

USENIX

c o n f e r e n c e

proceedings

Proceedings of the 14th Systems Administration Conference (LISA 2000)

New Orleans, LA, USA

December 2000

**14th Systems
Administration
Conference
(LISA 2000)**

*New Orleans, Louisiana
December 3–8, 2000*

Co-Sponsored by **The USENIX Association** and
SAGE, the System Administrators Guild

USENIX **SAGE**
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION THE SYSTEM ADMINISTRATORS GUILD

For additional copies of these proceedings contact

USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710 USA
Telephone: 510-528-8649
<http://www.usenix.org>
<office@usenix.org>

The price is \$35 for members and \$45 for nonmembers.

Past USENIX Large Installation Systems Administration Workshop
and Conference Proceedings (price: member/nonmember)

Systems Administration VI Conference	1992	Long Beach, CA	\$23/\$30
Systems Administration VII Conference	1993	Monterey, CA	\$25/\$33
Systems Administration VIII Conference	1994	San Diego, CA	\$22/\$29
Systems Administration IX Conference	1995	Monterey, CA	\$30/\$38
Systems Administration X Conference	1996	Chicago, IL	\$30/\$38
Systems Administration XI Conference	1997	San Diego, CA	\$30/\$38
Systems Administration XII Conference	1998	Boston, MA	\$32/\$40
Systems Administration XIII Conference	1999	Seattle, WA	\$32/\$40

Outside the U.S.A. and Canada, please add \$12
per copy for postage (via air printed matter); \$13 for LISA XIV.

Copyright © 2000 by The USENIX Association. All rights reserved.

This volume is published as a collective work.

Rights to individual papers remain with the author or the author's employer.

Permission is granted for the noncommercial reproduction
of the complete work for educational or research purposes.

ISBN 1-880446-13-8

AIX and AS/400 are trademarks of IBM, Inc.

CRYPTOCARD is a registered trademark of Cryptocard, Inc.

ConsoleServer 3200 is a trademark of Lightwave Communications, Inc.

Corebuilder 9000 is a trademark of 3Com, Inc.

DCE and DFS are trademarks of TransArc, Inc.

Entrust is a trademark of Entrust Technologies, Inc.

GroupWise and IPX are trademarks of Novell, Inc.

IRIS and UNICOS are trademarks of Silicon Graphics, Inc.

Linux is a trademark of Linus Torvalds.

Network Flight Recorder is a trademark of Network Flight Recorder, Inc.

Oracle is a trademark of Oracle Corporation.

SecurID is a trademark of RSA Security, Inc.

Solaris is a registered trademark of Sun Microsystems.

Sybase is a trademark of Sybase, Inc.

Tru64 is a trademark of Compaq.

UNIX is a registered trademark of Unix International.

WANPIPE is a trademark of Sangoma Technologies Inc.

Windows 2000 and Windows NT are registered trademarks of Microsoft, Inc.

USENIX acknowledges all trademarks appearing herein.

 Printed in the United States of America on 50% recycled paper, 10-15% post consumer waste.

USENIX Association

**Proceedings of the Fourteenth
Systems Administration Conference
(LISA XIV)**

**December 3-8, 2000
New Orleans, LA, USA**

TABLE OF CONTENTS

Author Index	v
Acknowledgments	vi
Preface	vii

Opening Remarks

Wednesday (9:00-10:30 am)

Chairs: Phil Scarr and Rémy Evard

GENERAL TRACK

Deep Thoughts

Wednesday (11:00-12:30)

Chair: Rémy Evard

Theoretical System Administration	1
<i>Mark Burgess, Oslo University College</i>	
An Expectant Chat about Script Maturity	15
<i>Dr. Alva L. Couch, Tufts University</i>	
An Improved Approach for Generating Configuration Files from a Database	29
<i>Jon Finke, Rensselaer Polytechnic Institute</i>	

You, A Rock, and A Hard Place

Wednesday (2:30-3:30)

Chair: Christine Hogan

Fokstraut and Samba – Dealing with Authentication and Performance Issues On A Large Scale Samba Service	39
<i>Robert Beck & Steve Holstead, University of Alberta</i>	
Designing a Data Center Instrumentation System	43
<i>Bob Drzyzgula, Federal Reserve Board</i>	
Improving Availability in VERITAS Environments	59
<i>Karl Larson, Tellme Networks; Todd Stansell, GNAC</i>	

Users and Passwords and Scripts, Oh My!

Wednesday (4:00-5:30)

Chair: Trey Harris

User-Centric Account Management and Heterogeneous Password Changing	67
<i>Doug Hughes, Auburn University</i>	
Pelendur: Steward of the Sysadmin	77
<i>Matt Curtin, Interhack Corporation; Sandy Farrar & Tami King, The Ohio State University</i>	
Network Information Management and Distribution in a Heterogeneous and Decentralized Enterprise Environment	85
<i>Alexander Kent & James Clifford, Los Alamos National Laboratory</i>	

The Toolshed

Thursday (11:00-12:30)

Chair: Phil Scarr

xps: Dynamic Tree Watching under X	95
<i>Rocky Bernstein, Breakaway Solutions</i>	
Extending UNIX System Logging with SHARP	101
<i>Matt Bing & Carl Erickson, Grand Valley State University</i>	
Peep (The Network Auralizer): Monitoring Your Network With Sound	109
<i>Michael Gilfix & Prof. Alva Couch, Tufts University</i>	

1984

Thursday (2:00-3:30)

Chair: Jeff Allen

Thresh – a Data-Directed SNMP Threshold Poller	119
<i>John Sellens, Certainty Solutions Inc.</i>	
eEMU: A Practical Tool and Language for System Monitoring and Event Management	131
<i>Jarra Voleynik, eEMUconcept Pty Ltd</i>	
Aberrant Behavior Detection in Time Series for Network Monitoring	139
<i>Jake D. Brutlag, WebTV</i>	

The Sorcerer's Apprentice

Thursday (4:00-5:30)

Chair: Josh Simon

PIKT: Problem Informant/Killer Tool	147
<i>Robert Osterlund, University of Chicago</i>	
Relieving the Burden of System Administration through Support Automation	167
<i>Allan Miller & Alex Donnini, HandsFree Networks</i>	
FTP Mirror Tracker: A Few Steps towards URN	181
<i>Alexei Novikov, Institute of Theoretical and Experimental Physics, Moscow, Russia; Martin Hamilton, Loughborough University, UK</i>	

Fully Automatic

Friday (9:00-10:30)

Chair: John Orthoefer

Deployme: Tellme's Package Management and Deployment System	187
<i>Kyle Oppenheim & Patrick McCormick, Tellme Networks</i>	
Automating Request-based Software Distribution	197
<i>Chris Hemmerich, Indiana University</i>	
Use of Cfengine for Automated, Multi-Platform Software and Patch Distribution	207
<i>David Ressman & John Valdés, University of Chicago</i>	

Building Blocks

Friday (11:00-12:30)

Chair: Ruth Milner

Unleashing the Power of JumpStart: A New Technique for Disaster Recovery, Cloning, or Snapshotting a Solaris System	219
<i>Lee "Leonardo" Amatangelo, Collective Technologies</i>	
A Linux Appliance Construction Set	229
<i>Michael W. Shaffner, Agilent Laboratories</i>	
Automating Dual Boot (Linux and NT) Installations	245
<i>Rajeev Agrawala, Rob Fulmer, & Shaun Erickson, Lucent/Bell-Labs Research</i>	

NETWORK TRACK

Analyze This!

Wednesday (2:00-3:30)

Chair: William LeFebvre

Wide Area Network Packet Capture and Analysis	255
<i>Jon Meek, American Home Products Corporation</i>	
Sequencing of Configuration Operations for IP Networks	265
<i>P. Krishnan, ISPsoft, Inc.; Tejas Naik, Bell Laboratories; Ganesan Ramu, CoSine Comm., Inc.; Roshan Sequeira, ISPsoft, Inc.</i>	
ND: A Comprehensive Network Administration and Analysis Tool	275
<i>Ellen L. Mitchell, Eric Nelson, & David K. Hess, Texas A&M University</i>	

Go With the Netflow

Wednesday (4:00-5:30)

Chair: David Williamson

Combining Cisco NetFlow Exports with Relational Database Technology for Usage Statistics, Intrusion Detection, and Network Forensics	285
<i>John-Paul Navarro, Bill Nickless, and Linda Winkler, Argonne National Laboratory</i>	
The OSU Flow-tools Package and Cisco NetFlow Logs	291
<i>Mark Fullmer, OARnet; Steve Romig, The Ohio State University</i>	
FlowScan: A Network Traffic Flow Reporting and Visualization Tool	305
<i>Dave Plonka, University of Wisconsin-Madison</i>	

Someone's Knocking at the Door?

Friday (11:00-12:30)

Chair: Cat Okita

Tracing Anonymous Packets to Their Approximate Source	319
<i>Hal Burch, Carnegie Mellon University; Bill Cheswick, Lumeta Corp.</i>	
Analyzing Distributed Denial Of Service Tools: The Shaft Case	329
<i>Sven Dietrich, NASA Goddard Space Flight Center; Neil Long, Oxford University; David Dittrich, University of Washington</i>	

... Don't Let Them In

Friday (2:00-3:30)

Chair: Simon Cooper

YASSP! A Tool for Improving Solaris Security	341
<i>Jean Chouanard, Xerox – Palo Alto Research Center</i>	
SubDomain: Parsimonious Server Security	355
<i>Crispin Cowan, Steve Beattie, Greg Kroah-Hartman, Calton Pu, Perry Wagle and Virgil Gligor, WireX Communications, Inc.</i>	
NOOSE – Networked Object-Oriented Security Examiner	369
<i>Bruce Barnett, General Electric Corporate Research & Development</i>	

AUTHOR INDEX

Rajeev Agrawala	245	Steve Holstead	39
Lee "Leonardo" Amatangelo	219	Doug Hughes	67
Bruce Barnett	369	Alexander Kent	85
Steve Beattie	355	Tami King	77
Robert Beck	39	P. Krishnan	265
Rocky Bernstein	95	Greg Kroah-Hartman	355
Matt Bing	101	Karl Larson	59
Jake D. Brutlag	139	Neil Long	329
Hal Burch	319	Patrick McCormick	187
Mark Burgess	1	Jon Meek	255
Bill Cheswick	319	Allan Miller	167
Jean Chouanard	341	Ellen L. Mitchell	275
James Clifford	85	Tejas Naik	265
Dr. Alva L. Couch	15	John-Paul Navarro	285
Dr. Alva L. Couch	109	Eric Nelson	275
Crispin Cowan	355	Bill Nickless	285
Matt Curtin	77	Alexei Novikov	181
Sven Dietrich	329	Kyle Oppenheim	187
David Dittrich	329	Robert Osterlund	147
Alex Donnini	167	Dave Plonka	305
Bob Drzyzgula	43	Calton Pu	355
Carl Erickson	101	Ganesan Ramu	265
Shaun Erickson	245	David Ressman	207
Sandy Farrar	77	Steve Romig	291
Jon Finke	29	John Sellens	119
Mark Fullmer	291	Roshan Sequeira	265
Rob Fulmer	245	Michael W. Shaffer	229
Michael Gilfix	109	Todd Stansell	59
Virgil Gligor	355	John Valdés	207
Martin Hamilton	181	Jarra Voleynik	131
Chris Hemmerich	197	Perry Wagle	355
David K. Hess	275	Linda Winkler	285

ACKNOWLEDGMENTS

PROGRAM CHAIRS

Rémy Evard, *Argonne National Labs*
Phil Scarr, *Certainty Solutions*

PROGRAM COMMITTEE

Jeff Allen, *WebTV Networks, Inc.*
David Blank-Edelman, *Northeastern University*
Strata Chalup, *VirtualNet Consulting*
Trey Harris, *Mail.com*
Christine Hogan, *Imperial College*
Doug Hughes, *Global Crossing*
Ruth Milner, *Nat'l Radio Astron. Observ.*
John Orthoefer, *HarvardNet*
David Parter, *University of Wisconsin*
John Sellens, *Certainty Solutions*
Josh Simon, *Collective Technologies*

READERS

William Annis, *UW Madison*
Erez Zadok, *Columbia University*
Kathy Penn, *University of Maryland*
Douglas Kingston, *Deutsche Bank*

INVITED TALKS COORDINATORS

Tom Limoncelli, *Lumeta Corporation*
Pat Wilson, *Dartmouth College*

NETWORK TRACK

William LeFebvre, *CNN Internet Technologies*
David Williamson, *Certainty Solutions*

SECURITY TRACK

Simon Cooper, *SGI*
Cat Okita, *Global Crossing*

WORK-IN-PROGRESS COORDINATOR

Peg Schafer, *Harvard University*

ADVANCED TOPICS WORKSHOP

Adam Moskowitz, *LION Biosciences Res.*

VERY LARGE STORAGE WORKSHOP

Stephen M. DiPietro, *Compaq Computer*

LARGE PROJECT WORKSHOP

Joel Avery, *Nortel Networks*

LARGE PROVIDERS WORKSHOP

John Orthoefer, *HarvardNet*
Cat Okita, *Global Crossing*

METALISA WORKSHOP

Tom Limoncelli, *Lumeta Corporation*
Cat Okita, *Global Crossing*

AFS WORKSHOP

Esther Filderman, *Pittsburgh Supercomp'g Ctr.*
Ted McCabe, *MIT*
Derrick Brashear, *Carnegie Mellon U.*

PROCESS/BENCHMARKING WKSHOP

Carolyn Hennings, *Megapipe*
Tom Limoncelli, *Lumeta Corporation*
Alva Couch, *Tufts University*

TEACHING SYSADMIN WORKSHOP

Curt Freeland, *Notre Dame*
John Sechrest, *PEAK, Inc.*

GURU-IS-IN COORDINATOR

Lee Damon, *amazon.com*

TERMINAL ROOM COORDINATOR

Lynda McGinley, *University of Chicago*

PROCEEDINGS PRODUCTION

Rob Kolstad, *Delos*

PREFACE

Bienvenue vers la Nouvelle-Orléans! Welcome to New Orleans! We've got quite a week planned for you here in the Big Easy. Our program includes 34 refereed papers, some of which are the best we've ever seen at a LISA conference. We've got the Invited Talks track again this year. And we welcome two new tracks to the LISA family this year: one on Security Administration and one on Network Administration.

We'd like to extend special thanks to Ellie Young who has been responsible for coordinating and managing this entire event. Thanks again to Dan Klein for the excellent tutorial suite that make LISA the premier system administration learning event.

Rob Kolstad brought his considerable typesetting skills on-board to produce the proceedings this year, and it was quite a job. This is the largest and most comprehensive set of proceedings ever produced for a LISA conference: 378 pages of papers.

Tom Limoncelli and Pat Wilson organized this year's Invited Talks track where an "embarrassment of riches" has produced the best Invited Talks track in years.

Bill LeFebvre and David Williamson assembled the new Network Administration track with some really outstanding content for network administrators.

Simon Cooper and Cat Okita coordinated the new Security Administration track. As security become ever-more important to the system administrator community, this track will provide a focused look at the state of security administration.

Lee Damon did an outstanding job coordinating the Guru-Is-In track. He and his gurus are available to help solve those quirky and difficult problems.

Finally, this conference would not be possible without the considerable talents and effort from the USENIX Office and the USENIX Conference Office. Thanks to them!

If you have any questions or problems during the conference, please don't hesitate to grab one of the conference organizers and let us know.

Have a great week in New Orleans!

Laissez les jeux commencer!

Your hosts,

Rémy Evard
Phil Scarr

Theoretical System Administration

Mark Burgess – Oslo University College

ABSTRACT

In order to develop system administration strategies which can best achieve organizations' goals, impartial methods of analysis need to be applied, based on the best information available about needs and user practices. This paper draws together several threads of earlier research to propose an analytical method for evaluating system administration policies, using statistical dynamics and the theory of games.

Introduction

System administration includes the planning, configuration and maintenance of computer systems. The discipline of system administration is traditionally founded on the anecdotal experiences of system managers [1, 2], but this can only be carried so far; formal (mathematical) analyses of system administration have only recently begun to enable more scientific studies to be carried out [3, 4]. A lack of formal methods makes it difficult to express objective truths about the field, avoiding marketing assertions and the vested interests of companies and individuals. The aim of this paper is to summarize a mathematical formulation of system administration, which can account for a basis of empirical evidence, and which provides an objective approach to study. This is central to the present discussion on developing system administration as a formal discipline.

In previous work by the author and collaborators, it has been shown how aspects of the average empirical behaviour of systems of computers and users can be modelled using fairly straightforward statistical ideas [5, 6, 7, 8]. This has allowed a coarse statistical model of computer systems to be built, which can be used as a backdrop for studies of system administration. Previous work by other authors has also attempted to look at computer ecosystems in terms of differential equations [9]. In the future additional mathematical models will, no doubt, be devised in order to study other issues.

One of the obstacles to formulating a complete theory of system administration is the complexity of interaction between humans and computers. There are many variables in a computer system, which are controlled at distributed locations. Computer systems are *complex* in the sense of having many embedded causal relationships and controlling parameters. Computer behaviour is strongly affected by human social behaviour, and this is often unpredictable. However, the central question in any scientific investigation is one of balance: can one formulate a quantitative theory of system administration, which is general enough to be widely applicable, but which is specific enough to admit analysis?

The outline of this paper is as follows. To begin the discussion some simplified assumptions about the aims of system administration are stated. Next, two types of quantitative model, describing a computer system interacting with users (possibly via a network), are described and the primitive operations which can be carried out within the scope of the models are identified. The two types of model are referred to as type I (passive) and type II (strategic). A method of quantifying the benefits and flaws of different strategies emerges from this discussion. Strategies for system administration and for user behaviour may then be formulated and arranged in a matrix allowing the task of administering a computer system can be described in precise game theoretical terms. The primary goal of this work is to provide the recipe for performing this kind of analysis.

Basic Assumptions

Capturing such a complex pursuit as system administration in a few simple rules is presumptuous, but approximately possible if one focuses on core activities. In order to make progress one must agree on some specific aims for users and administrators. The purpose of defining the aims of the interested parties in a computer system, is to come up with a good enough abstraction for system management that specific issues may be addressed in quantifiable terms. In this paper, the word 'system' will be taken to mean any organized collection of computers interacting with a group of users. The assumptions used are these:

- The aim of the system administrator is to keep the system alive and running well so that users can perform a maximum amount of useful work.
- The aim of benign users is to produce useful work using the system. This consumes resources.
- The aim of malicious users is to maximize their control over system resources.

A possible quantitative definition of 'useful work' is the amount of user-data modified on the computer system, plus the information transmitted to or from remote locations. This can be refined for specific purposes. Time spent fighting for control of a

damaged machine, or other users, for example, is not useful work for normal users.

This short list of aims does not encompass every eventuality, but it establishes a starting point. In addition to these points, it is necessary to provide a scheme of values about what is subjectively good or bad about the system. When are things going well and when are things going badly? This is done by specifying a *system policy* [10, 11].

A system policy is a specification of a system's configuration and its acceptable patterns of usage. A complete policy therefore affects the basic installation of the system and also the way it changes in time due to interaction with users.

A system policy is the pillar of truth and measuring stick against which one determines whether system activity is acceptable or unacceptable. A sufficiently complete system policy can also include a complete configuration blueprint and thus determine whether the state of configuration is acceptable. The central theorem, which was found in [3] is:

A sufficiently complete specification of a system policy leads to the notion of an ideal average state for the system. Over time, the ideal average state of the system degrades. The aim of system administration is to keep the system as close to its ideal state as possible.

The meaning of the theorem is that it identifies system administration as a *regulatory procedure*. This idea of regulatory action was originally introduced, using the term *convergence*, in connection with the system administration tool cfengine [5, 12, 13]. Cfengine is a program used to automate the regulation of host state, by making it converge towards its 'ideal state' with every execution of cfengine. For cfengine, the ideal state is achieved when every detail of a computer configuration appears to be correctly implemented and no changes to the system made by users contravene system policy. Thus 'state' refers to adherence to a policy. The cfengine model turns out to be a useful starting point for discussing system administration, since it offers a detailed and concrete idealization of system administration tasks in terms of sequences of primitive actions. Currently, cfengine does not have a complete picture of system state, at all levels, though part of the aim of this kind of work is to improve on that situation. However, by basing a study on this idea one also obtains, as a side effect, a theoretical evaluation of the model which can be used to improve cfengine's design in the future.

Policy, State, and Convergence

Without proving the central theorem in this paper, it is helpful to provide a brief explanation of how the ideal state is constructed, and why it is only possible to insist on an average description of idealness.

State is a snapshot of the condition of a system, which results from its current configuration and the history of all tasks which have consumed and released its resources over time. To picture state, it is helpful to think of a human analogy. In [6], the analogy with human health was drawn. Using another analogy, that of evolutionary fitness or adaptation for a purpose, host state can be envisioned on an arbitrary scale, which makes the ideal state that condition in which the system is best able to perform its tasks. As with humans, general fitness of a computer system is a combination of two parts: a part which is determined by inherited properties and a part which is the result of its interaction with the environment.

For humans, the state of fitness would be the sum of genetically determined attributes (roughly speaking, a policy for the operation of the organism) and current physical fitness (the attunement or degradation resulting from interaction with environment). For a computer system, there is a similar duality: state refers to a part which is the sum of all configuration and policy decisions (basic design quality) and a part which results from an interaction with users and network impulses (input/output).

Thus state separates into policy and environment. The state of a computer system $S(t)$ changes continuously with time, due mainly to the interaction with the environment, but also internally, as a general consequence of the second law of thermodynamics (a statistical inference which notes that the number of ways in which a system can be disordered is far greater than the number of ways it can be ordered (adhering to policy), thus any random change is statistically more likely to lead to disorder than to order). This is the principle of increase of entropy [14, 15].

The environment of a computer system can be thought of as an external batch of transactions (see Figure 1), i.e., input and output which appears and disappears as users interact with the system. Each transaction makes use of resources and has the possibility of affecting the state $S(t)$ of the system by an amount $\delta S(t)$. The number of transactions is generally large for periods of time over which one expects the host state to change significantly: transactions last usually milliseconds, whereas host behaviour is self-similar often over days and weeks [7].

From empirical studies [7, 8], one has a picture of a computer system as having an stable average condition over periods of time, but fluctuating considerably in response to specific transactions. In other words, over days and weeks, computer resources change, but over many weeks the pattern of usage has a mean value which shows a stable pattern. At any given time, the actual values of resource variables are different from the mean, but these differences average out to an average condition, or state. If this average state of the host is to be maintained near its ideal state then one hopes that the fluctuations from the mean

$\delta S(t)$ are small, i.e., the disturbance to the system resulting from user interactions results in only a small change to the actual state.

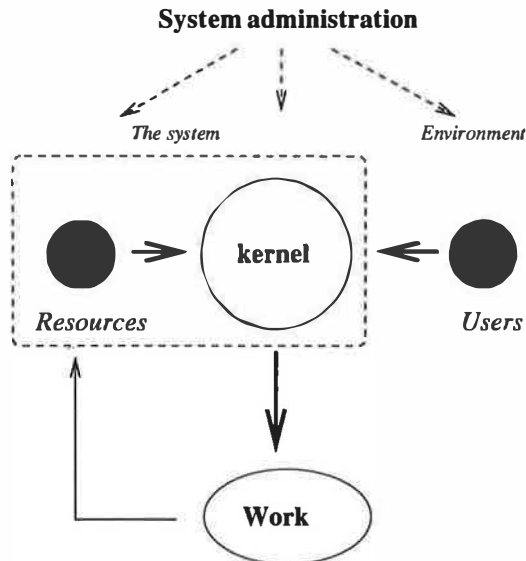


Figure 1: System administration is a regulative function over time.

There are thus four ideas of state which need to be considered: the ideal average state $\bar{S}^*(t)$, whose existence is implied by policy, the actual average state of the system $\bar{S}(t)$ which is the mean value of behaviour over weeks, the actual state of the system $S(t)$ and fluctuations from the average state $\delta S(t)$. All of these are functions of time. The latter three are related by a time-dependent relation:

$$S(t) = \bar{S}(t) + \delta S(t).$$

In order to speak of an average state, one has to say what average means. This has been defined precisely, using the theory of dynamical systems in [4] and turns out to be a regular arithmetical mean, calculated from a sliding window data sample, which advances over time. Because computer behaviour shows approximate periodic repetition, the average is defined in terms of co-cycles, over days and weeks: the two sociological influences which have profound implications for computer behaviour. Our empirical studies, at Oslo, have shown that fluctuations in the state must be averaged over a window of at least two or three weeks [7, 8] in order to see reliable stable behaviour under normal usage. This is the time scale over which users repeat their behaviour several times, within the framework of daily/weekly cycles.

Note that the ideal average state is only approximately constant (it changes slowly, at the same rate as the average changes), whereas the other states change with more rapidly time (on the time scale of individual interactions with the system). The average ideal state changes much more slowly than the actual state, precisely because it is averaged over coarser grains of time.

A notion of *idealness* can thus only be characterized for an average state because the system is constantly changing as users interact with it. Even the mean value is changing slowly with time. In physics of statistical systems, this is referred to as non-equilibrium behaviour. However, the fact that this decomposition is possible is important. It separates the effects of independent scales from one another. What happens in the short term is different to what happens on average, since one deviation might correct another, leaving no net problem. For example, if a user consumes a large amount of resources for a brief time (a temporary file, for instance) while performing useful work, this will only affect the actual state of the system for the duration of that task, provided the file is removed afterwards. A policy of file temporary file garbage collection can always remove such a file even if a user doesn't. The average state will therefore be relatively unaffected by short term changes. The meaning of the environmental ideal average state is therefore to define an interval of stability for interaction with environment. The system will always deviate from the so-called ideal state, but that need not be a problem as long as it does not deviate far from it for long periods of time.

There are two types of disturbance δS : those which (on average) preserve the state of the system, i.e., those which release as many resources as they consume, and those which consume resources without releasing them. The latter kind of disturbance is the most dangerous to the integrity of the system, since it can lead to runaway behaviour which sees the end of the system. This is the case in which it is necessary to introduce countermeasures to protect the state of the system. This is the purpose of computer immunology [6]. When large fluctuations are at hand, the system is in an intrinsically unstable state.

How can changes of state be characterized precisely? An obvious choice is to use the mathematical idea of a lattice, or discrete vector space. Although computer behaviour often has the appearance of a continuously varying load, the actual changes are all discrete in nature. Any interactive change in the system may be broken down into a sequence of discrete primitive operations. See Table 1 for the primitives used by cfengine [5, 12].

Any change to the configurable system, can be expressed in terms of these primitives. In addition to these, there are kernel variables which contain data that can be used to determine environmental state. Each independent primitive can be thought of as an axis in an n -dimensional lattice. Each change in the state of the system, of a given type, is a movement through the lattice in that direction. Moreover, since the averaging procedure for environment effectively divides up time into co-cyclic discrete units, (days and weeks) and scaled coarse-grained intervals, it is possible to draw the the state of the system on a lattice (see Figure 2). Mathematicians note: the lattice is only

conformally distorted by changes in the averaging procedure; the structure is preserved.

Primitive type T'	Comments/Examples
Create file object	Touch
Delete file object	Tidy garbage
Rename file object	Disable
Edit file	Configuration
Edit access control	Permissions
Request resource	Read/Mount
Copy file	Read/write
Process control	Start/stop
Process priority	Nice
Configure device	ifconfig/iocctl

Table 1: cfengine primitives.

In principle, there is a single point in the lattice which represents (at any given time) the most ideal state possible. In practice, one is only interested in keeping the system in a region, not too far from this practically unobtainable ideal point.

Changes to system policy must also be discrete strings of these primitives, since they have to be

implemented using the primitives, and thus a change in policy is simply a translation of the ideal state through the lattice.

Suppose one places the ideal state arbitrarily at the origin of this lattice. The further the system deviates from this origin, the more precarious the state of the system. Eventually when the state strays a sufficient distance from the origin, the system will exhaust its resources and fail completely. In the intervening distance, the system is working in accordance with policy when it is close to the ideal state. Using the Euclidean distance as a Hamming distance, for change in the system, it is possible to see that the number of corrective actions for required to return the system to its ideal state grows only linearly, however the number of possible corrective procedures increases exponentially.

The number of equivalent paths $H(\vec{d})$ back to the ideal state is:

$$H(\vec{d}) = \frac{(\sum_{j=1}^n d_j)!}{\prod_{k=1}^n (d_k!)}$$

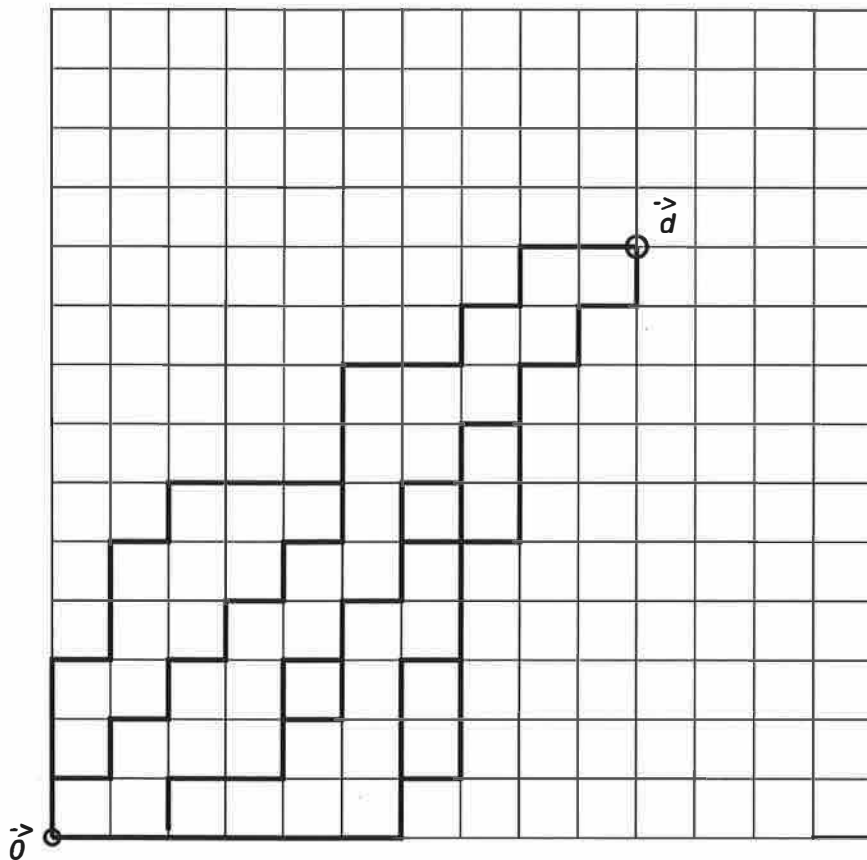


Figure 2: Deviations from the ideal state may be visualized as a random walk through a lattice of n -dimensions (here only two are shown). The number of paths of equal length by which one can return to the origin increases rapidly with the distance. For simplicity one may think of the axes as deviation due to policy and deviation due to usage.

This grows rapidly with the Euclidean distance $|\vec{d}|$:

$$|\vec{d}| \equiv d = \sqrt{\sum_{i=1}^n d_i^2}$$

The conclusion is that it is in the system's best interests to remain close to the ideal state at all times. If the remedy to a particular large deviation were unknown, the search for a remedy, in state space, would become extremely time-consuming as the magnitude of the problem increased. The expense or 'hopelessness' $H(\vec{d})$ measures this more than exponentially divergent problem. This hopeless search is the fate of any immune system without specific expert knowledge.

To summarize, every computer system, with a system policy, has an ideal state which is based on policy and environmental considerations. This ideal state fluctuates and degrades with time. The aim of system administration is to regulate the system to be close to this ideal state.

Modelling Computer Behaviour

An analysis of behaviour in a computer system, interacting with its environment of users and network impulses, requires two types of model, which may be referred to as passive (type I) and strategic (type II). The distinction refers to the level perceived intent behind the changes which take place.

In a type I description, the computer is viewed as being a mechanism coupled to a pseudo-periodic, random bath of impulses from an environment. One considers the effect of the this signal of impulses on the state $S(t)$. This is the view taken in [7, 8, 16, 4]. A type I model relates the behaviour of computers to that of other interacting, dynamical systems in mathematics and physics. This type of description is easily formulated and can be used to predict some of the average behaviour when the system is approximately stable. It works particularly well when large numbers of users, or transactions are involved, but not very well in situations of low usage. It is not good at predicting significant change, however, since the assumption of the method is that only gradual changes takes place over relatively long periods of time.

In a type II description, the computer system is viewed as the chequerboard for a game of competition between motivated individuals. This is the view taken in [3]. This type of analysis is designed to analyze the competitive processes which instigate significant change, at a more detailed level. It is good at determining the probability of success when using a set of strategies, and or finding the optimal strategies to solve a particular problem, but it is more difficult to apply than a type I description and relies on a knowledge or intuition of every relevant strategy which might be used by administrators and users alike.

Influences on the system can thus be classified as either random, stochastic or passive (type I), or as

intentional, adversarial or strategic (type II), depending on the significance of the change. This distinction is partly artificial: all changes can clearly be traced back to the actions of humans at some level, but it is not always useful to do this. Not all users act in response to an important provocation, or with a specific aim in mind. It just happens that their actions lead to a general average degradation of the ideal state, no malice intended. Thus there is a part of the spectrum of changes which averages out to a kind of faceless background noise: the details of who did what are of no concern [7, 8].

From type I models, based on the empirical studies made at Oslo, we have found that computer systems behave remarkably like photonic gases in the limit of long times [16]. That is, the occurrence of events on computer systems mimics the behaviour of black body radiation in physics. The interpretation of any dynamical variable as a fluctuating, statistical quantity is made possible by considering the effect of infinitesimal perturbation to dynamical variables $q(t)$. One begins by defining averages and correlated products of the fields $q(t)$, with action $S[q]$. The action is a generating functional which determines the constraints on the behaviour of the dynamical variable $q(t)$ by a variational principle. For the simplest dynamical systems, one may write

$$S = \int dV_t \frac{1}{2} q(t) \hat{O} q(t).$$

The sum over all fluctuations of given latency may be written [4]:

$$\Gamma[< q(t) >] = - \ln \int_{TB} d\mu e^{-S[< q > + \delta q]}.$$

The subscript TB refers stands for 'transaction bubbles' and refers to correlation graphs which are closed loops, i.e., complete transactions. This form is useful, since it is a self consistent form, which is derived on the assumption of linear statistical fluctuations an periodic time. It allows one to express self-consistent behaviour in terms of the measured variables alone. This quantity is essentially the free energy; it is a sum over all complete transactions in a fluctuating system and relies only on the assumed microscopic model which specifies available freedom and applied constraints. It can be calculated and compared to the fluctuation distributions measured for system variables.

Although the model is simple, the agreement with measured values is reasonable. The reason for this is a subtle but fascinating interaction between randomness and the order brought about by fixed daily and weekly rhythms. In fact, ensembles of events collected over weeks or months are insufficient in number to be perfectly described by these statistical methods, but the statistical model provides an idealized limit for computer behaviour, i.e., it provides a well defined envelope which approximates the system at scale of weeks. Moreover, it is so much simpler to understand than the actual behaviour of the system, that it has a valuable role to play in the discussion.

These studies have shown that computers behave like co-cyclic oscillators with periods of one day and one week (the rhythms imposed by the environment of users). Over periods of time which are long enough to gather enough data, these cyclic constraints reveal themselves as the shapes of distributions of events over time. They offer predictions about the statistical nature of the signal.

A type I model describes the average level of activity or *state* of the system which is related to the background noise. The second type of analysis which is required for a computer system is the analysis of non-cooperative user behaviour, i.e., analyzing which aspects of user behaviour affect the distribution of resources in the system. This analysis must be based on the *system policy*, since cooperation implies that the system is operating either within or outside the bounds of behaviour implied by the policy. Analysis must attempt to evaluate objectively the efficacy of different work patterns (strategies) employed by users in their interaction with the system.

A suitable framework for analyzing conflicts of interest, in a closed system, is the theory of games [17, 18]. Game theory is about introducing players, with goals and aims, into a scheme of rules and then analyzing how much a player can win, according to those restrictions. Each move in a game affords the player a characteristic value, often referred to as the 'payoff.' Game theory has been applied to warfare, to economics (commercial warfare) and many other situations. In this case, the game takes place on the n -dimensional board, spanned by the \vec{d} vectors.

Resource management is a problem of economics, just as energy flows in physical systems are to do with the economics of energy. The difference in system administration is only that there is no *a priori* currency for describing the economics of system administration. It is necessary to invent one. In social and economical systems one has money as the book-keeping parameter for transactions. In physical systems, one has energy as the book-keeping parameter. These quantities count resources, in some well-defined sense.

There are several types or classifications of game. Some games are trivial: one-person games of chance, for example, are not analyzable in terms of strategies, since the actions of the player are irrelevant to the outcome. In a sense, these are related to the first kind of model referred to above. Some situations in system administration fit this scenario. More interesting, is the case in which the outcome of the game can be determined by a specific choice of strategy on the part of the players. The most basic model for such a game is that of a two-person zero-sum game, or a game in which there are two players, and where the losses of one player are the gains of the other.

One feature which distinguishes the analysis proposed here from pure game theory is that the value

associated with different courses of action is not constant, but a function of time. The periodicities, discussed in the previous section must be taken into account as well as longer term changes, finite limits of system resources, non-linearities and so on. The implication of this is that the usefulness of a particular strategy varies according to when it is implemented.

The first kind of analysis assumes that the system has an average state and can therefore be used (at least in principle) to detect anomalous behaviour, e.g., behaviour which contravenes system policy. The second type of analysis looks at specific behavioural traits and attempts to evaluate their implications for the system state in more detail. Whether user behaviour lies within or outside the bounds of system policy is a matter of choice. Presumably one is interested in looking at all common behaviours, weighted by their likelihood in order to determine whether the system policy is effective enough. To make a type II theory realistic and tractable, one can imagine approximating the average background of the system activity using a type I model, and then studying specific strategies against this background. This leads to the notion of payoff, system currency in hybrid models.

Payoff in Type I and Type II Hybrids

Type I and type II models should not be should be thought of as completely separate issues: the best possible understanding of a computer system must involve both. Nonetheless it is primarily type II models which offer the chance to evaluate procedures and strategies of system administration. Type I models provide the background understanding of the resource behaviour, required to give substance to a type II model.

Equipped with a type I model for understanding the average interaction between user and system (which can be verified experimentally), one can construct a type II model in order to study a particular issue, against the average backdrop of type I activity. What is the outcome of introducing a new policy for governing a particular system resource, given what is known about how users generally interact with the system?

The determination of payoff, or the currency of the game is the central problem now. In order to find strategies which can keep the system close to its ideal state, one must assign a realistic value to strategies employed by users and system administrators. This is done by formulating a matrix (table) whose rows and columns specify the value or payoff associated with particular courses of action, for one of the players (see Figure 3). In the zero-sum approximation, it does not matter which player is chosen, since the losses of the one are the gains of the other. This is the only case to be considered here.

Courses of action available to each party, label the rows and columns. Rows are strategies and

columns are counter-strategies, or vice versa. The values within the matrix are the values gained by one of the players, in units of the arbitrary currency of the game when a given row-strategy and column-strategy are chosen. These values are determined by policy and by information about how resources behave, acquired from type I models: they are a set of value judgements about what is important or unimportant in the system and to what degree.

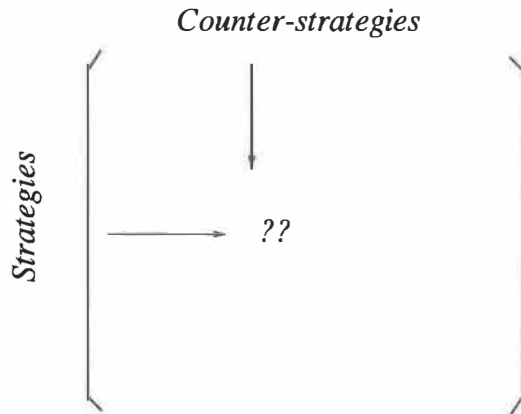


Figure 3: The payoff matrix is a table of strategies and counter strategies.

Once this 'payoff' matrix has been formulated, it contains information about the potential outcome of a game or scenario, using the strategies. This forms the basis for the theory of games [17, 18], whose methods and theorems make it possible to determine the optimal course or courses of action in order to maximize one's winnings. Obviously, any and all information which contributes to a judgement is useful, however one does not necessarily need a particularly detailed or accurate description to begin making simple value judgements about system behaviour. Even a simple quantification is useful, if it can distinguish between two possible courses of action.

How much can a user or an attacker hope to win? From our basic assumptions, the aim of a user is to maximize work produced or, in the worst case, maximize resources consumed. The system administrator, or embodiment of system policy, is not interested in winning the game for resources in the same way as users, but rather in confounding the game for users who gain too much control. The system administrator plays a similar role to that of a police force. In a vague sense, the administrator's job is to make sure that resources are distributed fairly, according to the policies laid down for the computer society (a Robin Hood role of altruistic government).

What is the currency of this evaluation? A definition is required in order to quantify the production of useful work by the system and its users. Clearly the term 'useful work' spans a wide variety of activities. Clearly work can increase and decrease (work can be lost through accidents), but this is not really germane

to the problem at hand. The work generated by a user (physical and mental work and then computationally assisted results) is a function of the information input into the system by the user. Since the amount of computation resulting from a single input might be infinite, in practice, the function is an unknown.

In addition to the actual work produced by a user's strategy, other things might be deemed to be of value, such as privilege and status. In a community, wealth does not guarantee privilege or status unless that coincides with the politics of the community. Pay-off can therefore be a complex issue to model. If one includes these ranking issues into calculations, one might allow for the possibility that a user plays the system rules in order to gain privileges for some later purpose. A user who accrues the goodwill of the system administrator, might eventually gain trust or even special privileges, such as extra disk space, access to restricted data etc. Such problems are of special interest in connection with security [19, 20].

For simplicity, the discussion of type II models in this paper refers only to games with two players. In a community, games are not necessarily two player zero sum engagements however. What is lost by one player is not necessarily gained by an obvious opponent. Moreover, the information available to different sides in a conflict can affect their modes of play. The so-called prisoner's dilemma, leads to the famous Nash equilibrium [21] which is a trade-off:

A user of the system who pursues solely private interests, does not necessarily promote the best interest of the community as a whole.

Should users cooperate or fight to maximize their winnings? Users can sabotage their own self-interest by using up all the available resources on a finite system, gaining enemies or losing the goodwill of system police. Strategies which succeed in encouraging users to comply with guidelines can therefore be an effective way of ensuring a fair use of resources. The main reason for considering two person games here is the overriding simplicity of the two person game, compared to including more players. This should not be taken to imply that more complex models will not be important.

In a realistic situation one expects both parties in the two-person game to use a mixture of strategies. The number of possible strategies is huge and the scope for strategic contrivance is almost infinite. Strategies can be broken down into linear combinations of primitives just as any operation on the system can. What then is a strategy?

- An array of operations
- A schedule for the operations
- Rules for counter-moves or responses

In addition to simple strategies, there can be meta-strategies, or long-term goals. For instance, a nominal community strategy might be to:

- Maximize productivity or generation of work.
- Gain the largest feasible share of resources.
- An attack strategy might be to
- Consume as many resources as possible.
- Destroy key resources.

Other strategies for attaining intermediate goals might include covert strategies such as *bluffing* (falsely naming files). Users can obey or ignore policy restrictions, use decoys, escalate or mitigate hostilities, attack/kill/delete a resource, retaliate. Defensive strategies might involve taking out an attacker, counter attacking, or evasion (concealment), exploitation, trickery, antagonization, incessant complaint (spam), revenge etc. Security and privilege, levels of access, integrity and trust must be woven into algebraic measures for the pay-off. One of the advantages of this formulation on system administration is that it places regular administration on the same footing as security issues. These were never separate issues and should not be considered as such, even in today's more security aware climate.

A means of expressing these devices must be formulated within a language which can be understood by system administrators, but which is primitive enough to enable the problem to be analyzed algebraically.

Example Games

The difficult part of a type II analysis is turning the high level concepts and aims listed above, into precise numerical values. To illustrate the procedure, consider an example of some importance, namely the filling of user disks. The need for forced garbage collection has been argued on several occasions [22, 5, 12], but the effectiveness of different strategies for avoiding disk may now be analyzed theoretically. This analysis is inspired by the user environment at Oslo University College, and the expressions derived here are designed to model this situation, not an arbitrary system.

The currency of this game must first be agreed upon. What value will be transferred from one player to the other in play? There are three relevant measurements to take into account: (i) the amount of resources consumed by the attacker (or freed by the defender); sociological rewards: (ii) 'goodwill' or (iii) 'privilege' which are conferred as a result of sticking to the

policy rules. These latter rewards can most easily be combined into an effective variable 'satisfaction.' A 'satisfaction' measure is needed in order to set limits on individuals' rewards for cheating, or balance the situation in which the system administrator prevents users from using any resources at all. This is clearly not a defensible use of the system, thus the system defenses should be penalized for restricting users too much. The characteristic matrix now has two contributions,

$$\pi = \pi_r(\text{resources}) + \pi_s(\text{satisfaction}) .$$

It is convenient to define

$$\pi_r \equiv \pi(\text{resources}) = \frac{1}{2} \left(\frac{\text{Resources won}}{\text{Total resources}} \right) .$$

Satisfaction π_s is assigned arbitrarily on a scale from plus to minus one half, such that, Satisfaction π_s is assigned arbitrarily on a scale from plus to minus one half,

$$\begin{aligned} -\frac{1}{2} &\leq \pi_r \leq +\frac{1}{2} \\ -\frac{1}{2} &\leq \pi_s \leq +\frac{1}{2} \\ -1 &\leq \pi \leq +1 . \end{aligned}$$

The pay-off is related to the movements made through the lattice \vec{d} . The different strategies can now be regarded as duels, or games of timing; see Table 2. These elements of the characteristic matrix must now be filled, using a model and a policy. A general expression for the rate at which users produce files is approximated by:

$$r_u = \frac{n_b r_b + n_g r_g}{n_b + n_g} ,$$

where r_b is the rate at which bad users (i.e., problem users) produce files, and r_g is the rate for good users. The total number of users $n_u = n_b + n_g$. From experience, the ratio $\frac{n_b}{n_g}$ is about one percent. The rate can be expressed as a scaled number between zero and one, for convenience, so that $r_b = 1 - r_g$.

The payoff in terms of the consumption of resources by users, to the users themselves, can then be modelled as a gradually accumulation of files, in daily waves, which are a maximum around midday:

$$\pi_u = \frac{1}{2} \int_0^T dt \frac{r_u(\sin(2\pi t/24) + 1)}{R_{\text{tot}}} ,$$

where the factor of 24 is the human daily rhythm, measured in hours, and R_{tot} is the total amount of

Users/System	Ask to tidy	Tidy by date	Tidy above Threshold	Quotas
Tidy when asked	$\pi(1,1)$	$\pi(1,2)$	$\pi(1,3)$	$\pi(1,4)$
Never tidy	$\pi(2,1)$	$\pi(2,2)$	$\pi(2,3)$	$\pi(2,4)$
Conceal files	$\pi(3,1)$	$\pi(3,2)$	$\pi(3,3)$	$\pi(3,4)$
Change timestamps	$\pi(4,1)$	$\pi(4,2)$	$\pi(4,3)$	$\pi(4,4)$

Table 2: Games of timing.

resources to be consumed. Note that, by considering only good user or bad users, one has a corresponding expression for π_g and π_b , with r_u replaced by r_g or r_b respectively. An automatic garbage collection system (cfengine) results in a negative pay-off to users, i.e., a pay-off to the system administrator. This may be written

$$\pi_a = \frac{1}{2} \int_0^T dt \frac{r_a(\sin(2\pi t/T_p) + 1)}{R_{tot}},$$

where T_p is the period of execution for the automatic system (in our case, cfengine). This is typically hourly or more often, so the frequency of the automatic cycle is some twenty times greater than that of the human cycle. The rate of resource-freeing r_a is also greater than r_u , since file deletion takes little time compared to file creation, and also an automated system will be faster than a human. The quota payoff yields a fixed allocation of resources, which are assumed to be distributed equally amongst users and thus each quota slice assumed to be unavailable to other users. The users are nonchalant, so $\pi_s = 0$ here, but the quota yields

$$\pi_q = + \frac{1}{2} \left(\frac{1}{n_h + n_g} \right).$$

The matrix elements are expressed in terms of these.

$\pi(1,1)$: Here $\pi_s = -1/2$ since the system administrator is as satisfied as possible by the users' behaviour. π_r is the rate of file creation by good users π_g , i.e., only legal files are produced. Comparing the strategies, it is clear that $\pi(1,1) = \pi(1,2) = \pi(1,3)$.

$\pi(1,4)$: Here $\pi_s = 0$ reflecting the users' dissatisfaction with the quotas, but the system administrator is penalized for restricting the freedom of the users. With fixed quotas, users cannot generate large temporary files. π_q is the fixed quota payoff, a fair slice of the resources. Clearly $\pi(4,1) = \pi(4,2) = \pi(4,3) = \pi(4,4)$. The game has a fixed value if this strategy is adopted by system administrators. However, it does not mean that this is the best strategy, according to the rules of the game, since the system administrator loses points for restrictive practices, which are not in the best interest of the organization. This is yet to be determined.

$\pi(2,1)$: Here $\pi_s = 1/2$ since the system administrator is maximally dissatisfied with users' refusal to tidy their files. The pay-off for users is also maximal in taking control of resources, since the system administrator does nothing to

prevent this, thus $\pi_r = \pi_u$. Examining the strategies, one finds that $\pi(2,1) = \pi(3,1) = \pi(3,2) = \pi(3,3) = \pi(4,1) = \pi(4,2)$.

$\pi(2,2)$: Here $\pi_s = 1/2$ since the system administrator is maximally dissatisfied with users' refusal to tidy their files. The pay-off for users is now mitigated by the action of the automatic system which works in competition, thus $\pi_r = \pi_u - \pi_a$. The automatic system is invalidated by user bluffing (file concealment).

$\pi(2,3)$: Here $\pi_s = 1/2$ since the system administrator is maximally dissatisfied with users' refusal to tidy their files. The pay-off for users is mitigated by the automatic system, but this does not activate until some threshold time is reached, i.e., until $t > t_0$. Since changing the date cannot conceal files from the automatic system, when they are tidied above threshold, we have $\pi(2,3) = \pi(4,3)$.

Thus, in summary, the characteristic matrix is given by Formula 1 where the step function is defined by,

$$\Theta(t_0 - t) = \begin{cases} 1 & (t \geq t_0) \\ 0 & (t < t_0) \end{cases},$$

and represents the time-delay in starting the automatic tidying system in the case of tidy-above-threshold. This was explained in more detail in [3].

It is possible to say several things about the relative sizes of these contributions. The automatic system works at least as fast as any human so, by design, in this simple model we have

$$\frac{1}{2} \geq |\pi_a| \geq |\pi_u| \geq |\pi_g| \geq 0,$$

for all times. For short times $\pi_q > \pi_u$, but users can quickly fill their quota and overtake this. In a zero-sum game, the automatic system can never tidy garbage faster than users can create it, so the first inequality is always saturated. From the nature of the cumulative pay-offs, we can also say that

$$\left(\frac{1}{2} + \pi_u\right) \geq \left(\frac{1}{2} + \pi_u + \pi_a \Theta(t_0 - t)\right) \geq \left(\frac{1}{2} + \pi_u + \pi_a\right),$$

and

$$\left|\frac{1}{2} + \pi_u\right| \geq |\pi_g - \frac{1}{2}|.$$

Applying these results to a modest strategy of automatic tidying, of garbage, referring to Figure 4, one sees that the automatic system can always match users' moves. As drawn, the daily ripples of the automatic system are in phase with the users' activity. This is not realistic, since tidying would normally be done

$$\pi(u, s) = \begin{pmatrix} -1/2 + \pi_g(t) & -1/2 + \pi_g(t) & -1/2 + \pi_g(t) & \pi_q \\ 1/2 + \pi_u(t) & 1/2 + \pi_u(t) + \pi_a(t) & 1/2 + \pi_u(t) + \pi_a(t)\Theta(t_0 - t) & \pi_q \\ 1/2 + \pi_u(t) & 1/2 + \pi_u(t) & 1/2 + \pi_u(t) & \pi_q \\ 1/2 + \pi_u(t) & 1/2 + \pi_u(t) & 1/2 + \pi_u(t) + \pi_a(t)\Theta(t_0 - t) & \pi_q \end{pmatrix}$$

Formula 1: Characteristic matrix.

at night when user activity is low, however such details need not concern us in this illustrative example.

The policy created in setting up the rules of play for the game, penalizes the system administrator for employing strict quotas which restrict their activities. Even so, users do not gain much from this, because quotas are constant for all time. A quota is a severe handicap to users in the game, except for very short times before users reach their quota limits. Quotas could be considered cheating by the system administrator, since they determine the final outcome even before play commences. There is no longer an adaptive allocation of resources. Users cannot create temporary files which exceed these hard and fast quotas. An immunity-type model which allows fluctuations is a more resource efficient strategy in this respect, since it allows users to span all the available resources for short periods of time, without consuming them for ever.

According to the *minimax* theorem, proven by John Von Neumann, any two-person zero-sum game has a solution, either in terms of a pair of optimal *pure* strategies or as a pair of optimal *mixed* strategies [17, 18]. The solution is found as the balance between one player's attempt to maximize his pay-off and the other player's attempt to minimize the opponent's result. In

general one can say of the pay-off matrix that

$$\max \min \pi_{rc} \leq \min \max \pi_{rc},$$

where the arrows refer to the directions of increasing rows (\downarrow) and columns (\rightarrow). The left hand side is the least users can hope to win (or conversely the most that the system administrator can hope to keep) and the right is the most users can hope to win (or conversely the least the system admin can hope to keep). If we have Equation 2,

$$\max \min \pi_{rc} = \min \max \pi_{rc}$$

Equation 2: Equality in the payoff matrix.

it implies the existence of a pair of single, pure strategies (r^* , c^*) which are optimal for both players, regardless of what the other does. If the equality is not satisfied, then the minimax theorem tells us that there exist optimal mixtures of strategies, where each player selects at random from a number of pure strategies with a certain probability weight.

The situation for our time-dependent example matrix is different for small t and for large t . The distinction depends on whether users have had time to exceed fixed quotas or not; thus 'small t ' refers to times when users are not impeded by the imposition of quotas. For small t , one has:

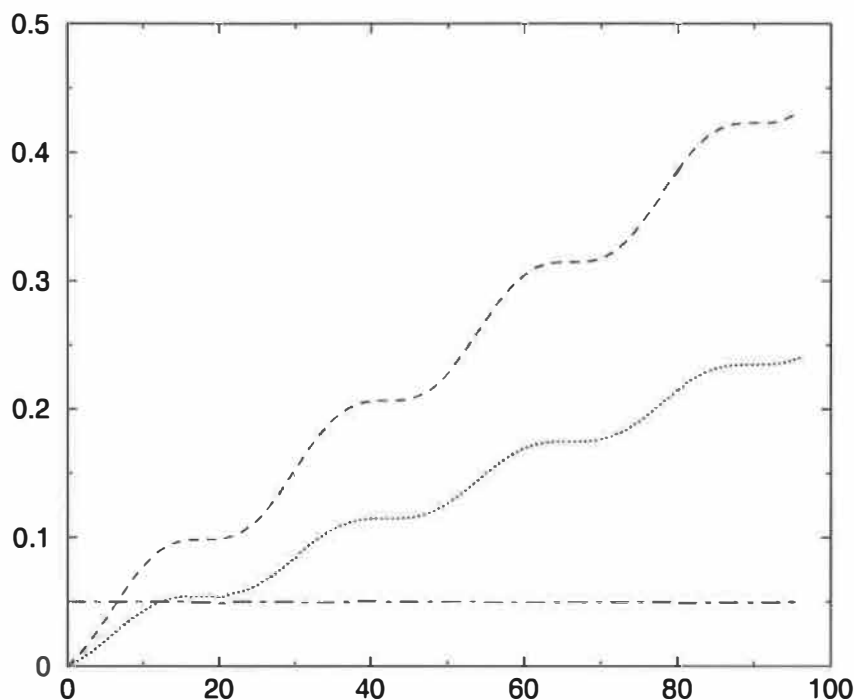


Figure 4: The absolute values of pay-off contributions as a function of time (in hours), For daily tidying $T_p = 24$. User numbers are set in the ratio $(n_g, n_b) = (99, 1)$, based on rough ratios from the author's College environment, i.e., one percent of users are considered mischievous. The filling rates are in the same ratio: $r_b/R_{tot} = 0.99$, $r_g/R_{tot} = 0.01$, $r_a/R_{tot} = 0.1$. The flat dot-slash line is $|\pi_q|$, the quota pay-off. The lower wavy line is the cumulative pay-off resulting from good users, while the upper line represents the pay-off from bad users. The upper line doubles as the magnitude of the pay-off $|\pi_a| \geq |\pi_u|$, if we apply the restriction that an automatic system can never win back more than users have already taken. Without this restriction, $|\pi_a|$ would be steeper.

$$\max_{\downarrow} \min_{\rightarrow} \pi_{rc} = \max_{\downarrow} \begin{pmatrix} \pi_g - \frac{1}{2} \\ \frac{1}{2} + \pi_u + \pi_a \\ \frac{1}{2} + \pi_u \\ \frac{1}{2} + \pi_u + \pi_a \Theta(t_0 - t) \end{pmatrix} = \frac{1}{2} + \pi_u.$$

The ordering of sizes in the above minimum vector is:

$$\frac{1}{2} + \pi_u \geq \frac{1}{2} + \pi_u + \pi_a \Theta(t_0 - t) \geq \pi_u + \pi_a \Theta(t_0 - t) \geq \pi_g - \frac{1}{2}$$

For the opponent's endeavours one has

$$\min_{\rightarrow} \max_{\downarrow} \pi_{rc} = \min_{\rightarrow} \left(\frac{1}{2} + \pi_u, \frac{1}{2} + \pi_u, \frac{1}{2} + \pi_u, \pi_q \right) = \frac{1}{2} \pi_u.$$

This indicates that the equality in Equation 2 is satisfied and there exists at least one pair of pure strategies which is optimal for both players. In this case, the pair is for users to conceal files, regardless of how the system administrator tidies files (the sysadm's strategies all contribute the same weight in Equation 2. Thus for small times, the users are always winning the game if one assumes that they are allowed to bluff by concealment. If the possibility of concealment or bluffing is removed (perhaps through an improved technology), then the next best strategy is for users to bluff by changing the date, assuming that the tidying looks at the date. In that case, the best system administrator strategy is to tidy indiscriminately at threshold.

For large times (when system resources are becoming or have become scarce), then the situation looks different. In this case one finds that

$$\max_{\downarrow} \min_{\rightarrow} \pi_{rc} = \min_{\downarrow} \max_{\rightarrow} \pi_{rc} = \pi_q.$$

In other words, the quota solution determines the outcome of the game for any user strategy. As already commented, this might be considered cheating or poor use of resources, at the very least. If one eliminates quotas from the game, then the results for small times hold also at large times.

This simple example of system administration as a strategic game between users and administrators is only an illustration of the principles involved in building a type I/II hybrid model. In spite of its simplicity, it is already clear that user bluffing and system quotas are strategies which are to be avoided in an efficient system. The value of 'goodwill' in curbing anti-social behaviour should not be underestimated. By following this basic plan, it should be possible to analyze more complex situations in future work.

Future Work

From the type I models studied at Oslo [4, 7], it appears that the most important characteristic of the average user behaviour is its periodicity: the average

state of computers has a daily period and a weekly period; these trace the social cycles of users all around the world. It is possible, as more is learned, that more detailed characteristics will emerge which are general enough to be used in a type I/type II hybrid. The main promise of type I theories lies in the possibility of anomaly detection and self-analysis, leading to fault detection, intrusion detection and improvements in immune system technology at the user level (e.g., cfengine). However, it is also important to know, for strategic analysis, when the system is most loaded, most vulnerable and most available.

Only one example of a type II theory has been examined here. What other issues might be studied by a type II model? The possibilities include strategies such as: consolidation versus distribution in system planning (where should resources be located?); delegation vs centralization; choosing many simple tools or a few powerful ones [23] (cost of learning and support, functionality, likelihood of bugs, results, rate of evolution of task and tools); the effect of system work ethics on productivity in a business (does the business spend most of its time working against itself or its competitors?); is the best strategy one which leads to stability or perfection? Mission critical systems and high security systems are obvious candidates for analysis. Other resources uses: network share, processes, setting of permissions, placement of security etc. The possibilities are limited only by the imagination. The benefit of the type II model is in setting up a systematic method for making impartial judgements about strategies for system management and system regulation.

A common theme in all strategic studies, involving complex competitive behaviour, is the so-called Red Queen scenario. This is about working hard to maintain the status quo; it is a reference to a scene from Alice Through The Looking Glass:

"Well in our country," said Alice, still panting a little, "you'd generally get to somewhere else – if you ran very fast for a long time as we've been doing."

"A slow sort of country!" said the Queen. "Now here, you see, it takes all the running you can do, to keep in the same place. If you want to get somewhere else, you must run twice as fast as that!"

This is also referred to as an arms race. In a true dynamical system, nothing stands still. Adapting one's strategies to be optimal over time means a continual reappraisal of their efficacy. One has to be running all the time to keep up with the environment, continually adapting to change. The need for garbage collection is an example of this.

In choosing strategies which walk the line between compliance and conflict, within a framework of rules (the prisoner's dilemma), some studies have indicated that the best solutions are often cooperation at first, and then *tit for tat* after that, if cooperation

does not work: i.e., try cooperation first to mitigate hostilities, and then send a message that one means business thereafter.

Sketching out this recipe for analyzing system administration policies reveals some potent ideas, which have a merit quite independently of their analytical value. The idea of using mixtures of strategies to most efficiently regulate a system, is so close to the ideas used in cfengine [5, 12] as to suggest that they could be adopted more even explicitly, and more dynamically. Rather than relying on batch operation, a policy engine like cfengine could be more dynamical in its responses to deviations from ideal state, and be able to set in motion a variety of parallel responses, which might extend its reach in dealing with more dynamical problems like network intrusions. It could also respond to more long-term trends in system usage and adapt its behaviour accordingly. Part of the motivation of this work was precisely to see what could be done to improve on cfengine [24]. Once refined, the approach in this paper will lead to improvements in cfengine, and improve the automation of host security.

Summary

The aim of this paper has been to propose a framework for analyzing models of system administration. Its main contention is that it is possible to see system administration as the effort to keep the system close to an ideal state, by introducing countermeasures in the face of competitive resource consumption. This is the formal basis which opens the way for objective analyses in the field.

With a mathematical approach, it becomes easier to see through personal opinions and vested interests when assumptions and methods are clearly and rigorously appraised. However, one can only distinguish between those possibilities which are taken into account. That means that every relevant strategy, or alternative, has to be considered. This is the limitation of game theory. It is not possible to determine strategies without the creative input of experts, and a clearly described policy.

Appealing only to a simple-minded analysis of disk filling, some straightforward conclusions are possible: the use of quotas is an inefficient way of counteracting the effects of selfish users, when the whole community's interests are taken into account. A quota strategy can never approach the same level of productivity as one which is based on competitive counterforce. The optimal strategies for garbage collection are rather found to lie in the realm of the immunity model [6, 15]. However, it is a sobering thought that a persistent user, who is able to bluff the immune system into disregarding it, (like a cancer) will always win against the resource battle. The need for new technologies which can see through bluffs will be an ever present reality in the future. With the ability of encryption and compression systems to obscure file contents, this is a

contest which will not be easily won by system administrators.

Author Information

Mark Burgess is an associate professor of physics and computer science at Oslo University College, creator of cfengine and author of the book *Principles of Network and System Administration*. He may be reached at mark@iu.hio.no or <http://www.iu.hio.no/~mark>. Cfengine can be obtained from <http://www.iu.hio.no/cfengine>. Oslo University College's research pages for system administration are at <http://www.iu.hio.no/SystemAdmin>.

References

- [1] R. Evard, "An Analysis of Unix System Configuration," *Proceedings of the 11th Systems Administration Conference (LISA)*, page 179, 1997.
- [2] S. Traugott and J. Huddleston, "Bootstrapping an Infrastructure," *Proceedings of the 12th Systems Administration Conference (LISA)*, page 181, 1998.
- [3] M. Burgess, "On the theory of system administration," Submitted to the *Journal of the ACM*, 2000.
- [4] M. Burgess, "Information theory and the kinematics of distributed computing, submitted to *Physical Review E*, 2000.
- [5] M. Burgess, "A site configuration engine," *Computing systems*, 8:309, 1995.
- [6] M. Burgess, "Computer immunology," *Proceedings of the 12th Systems Administration Conference (LISA)*, page 283, Usenix Association, 1998.
- [7] M. Burgess, H. Haugerud, and S. Straumsnes, "Measuring Host Normality, I," submitted to *Software Practice and Experience*, 1999.
- [8] M. Burgess and Trond Reitan, "Measuring Host Normality, II," submitted to *Software Practice and Experience*, 1999.
- [9] N. Glance, T. Hogg, and B. A. Huberman, "Computational Ecosystems in a Changing Environment," *International Journal of Modern Physics*, C2:735, 1991.
- [10] E. D. Zwicky, S. Simmons, and R. Dalton, "Policy as a system administration tool," *Proceedings of the Fourth Systems Administration Conference (LISA)*, SAGE/USENIX, page 115, 1990.
- [11] B. Howell and B. Satdeva, "We have met the enemy: An Informal Survey of Policy Practices in the Internetworked Community," *Proceedings of the fifth systems administration conference (LISA)*, SAGE/USENIX, page 159, 1991.
- [12] M. Burgess and R. Ralston, "Distributed resource administration using cfengine," *Software practice and experience*, 27:1083, 1997.

- [13] M. Burgess, "Automated system administration with feedback regulation," *Software Practice and Experience*, 28:1519, 1998.
- [14] F. Reif, *Fundamentals of Statistical Mechanics*, McGraw-Hill, Singapore, 1965.
- [15] M. Burgess, *Principles of Network and System Administration*, J. Wiley & Sons, Chichester, 2000.
- [16] M. Burgess, "Thermal, non-equilibrium phase space for networked computers," *Physical Review*, E62:(in press), 2000.
- [17] J. V. Neumann and O. Morgenstern, *Theory of games and economic behaviour*. Princeton University Press, Princeton, 1944.
- [18] M. Dresher, *The mathematics of games of strategy*, Dover, New York, 1961.
- [19] V. Jones and D. Schrodel, "Balancing security and convenience," *Proceedings of the First Systems Administration Conference (LISA)*, (SAGE/USENIX), page 5, 1987.
- [20] I. S. Winkler and B. Dealy, "Information Security Technology? Don't Rely On It. A Case Study in Social Engineering," *Proceedings of the 5th USENIX Security Symposium*, page 1, 1995.
- [21] J. F. Nash. *Essays on Game Theory*, Edward Elgar, Cheltenham, 1996.
- [22] E. D. Zwicky, "Disk space management without quotas," *Proceedings of the third systems administration conference (LISA)*, (SAGE/USENIX), page 41, 1989.
- [23] H. E. Harrison, "Maintaining a consistent software environment," *Proceedings of the First Systems Administration Conference (LISA)*, (SAGE/USENIX), page 16, 1987.
- [24] M. Burgess, "Evaluating cfengine's Immunity Model of System Maintenance, *Proceedings of USENIX/SANE 2000, Netherlands*, 2000.

An Expectant Chat about Script Maturity

Dr. Alva L. Couch – Tufts University

ABSTRACT

Using scripts to automate common administrative tasks is a ubiquitous practice. Powerful scripting languages and approaches support seemingly 'efficient' scripting practices that actually compromise the robustness of our scripts, as well as indirectly detracting from the stability and maturity of our support infrastructure. This is especially true for scripts that automate complex interactive processes using the scripting tools Expect or Chat. I present a formal methodology for the design and implementation of interactive scripting that, with a little more effort than writing a simple Expect script, produces scripts with substantially improved robustness and permanence. My scripting tool Babble interprets a detailed structural description of an interactive session as a script. Using this declarative, fourth-generation language, one can craft interactive scripts that are easier to perfect, inherently more robust, easier to maintain over time, and self-documenting.

Introduction

The amazing powers of current rapid prototyping languages strongly entice us to ease our burdens by writing simple scripts to automate repetitive administrative tasks. But the nature of our profession also encourages us to cut corners on these scripts, writing in haste to satisfy often inadequately predefined needs. Our scripts are not subjected to rigorous software engineering process or testing. They are easy to write but almost completely undocumented and difficult for anyone but the author to understand and maintain. The process of script writing is evolutionary rather than planned, driven by expediency rather than coherent overall design. This accelerates the 'software rot' that is unavoidable in all software development processes [3].

But even when we employ the best accepted software engineering process, writing system administration automation scripts is actually more difficult than writing many other types of software. Administrative scripts have strong couplings to their operating environment and make substantive changes to their environment as they execute. They are highly embedded [2] systems with complex preconditions and requirements for script success. An administrative script can be faced with any pre-existing conditions, be required to modify anything, and be required to produce most any result. And errors in a script executed with administrative powers can have dire results.

We can most easily understand the perils in writing scripts by considering the target system configuration as a collection of global variables. No one writes 'normal' software using global variables anymore, because of the danger of creating code that makes undocumented and untraceable changes in unpredictable places. But we do the equivalent in writing privileged administrative scripts on a day-to-day basis.

Scripts and Organizational Maturity

Scripts are not simply passive tools that we can use or ignore on a whim. Once deployed, they have an

active role in determining the maturity of our service organization as defined in the System Administration Maturity Model, or SAMM [18], based upon the Capabilities Maturity Model of software engineering [7, 29]. One goal of SAMM is to encourage stable organizational structures in which particular staff members are interchangeable and replaceable on a moment's notice. The scripts that we craft to ease our lives can violate this principle in a rather subtle way.

Ad-hoc scripts often possess hidden usability constraints and behaviors only known to the author. If they work when we use them, fine; else we page the author, who repairs the damage thus inflicted. It is easy for the rest of us to relax into complacency as long as the author responds to pages in a timely manner. But regardless of the benevolence and good intentions of the author, using such a script compromises the maturity of the whole service organization, because the author becomes an irreplaceable component and service bottleneck instead of being interchangeable with other staff.

Because of this effect, some site managers (who shall remain nameless) prohibit ad-hoc scripting and automation, so that anything that cannot be automated by high-quality, well-documented, industrial-strength automation tools is done entirely by hand, avoiding scripting wherever possible. I take the controversial stance that this seemingly strange decision is justifiable. By making this choice, all their staff remain interchangeable and replaceable on a moment's notice, thus increasing their support organization's stability and maturity, at the cost of reducing individual productivity.

I formed this controversial opinion from direct experience. I am not an average script writer. I have written over 30,000 lines of Perl in the ten years I have known the language, to achieve many different ends. But I have also had a unique opportunity to observe the impact of my own scripts upon operations

in my absence. When I 'retired' two years ago from technical to managerial duties, I left all of my 'clever' administrative scripts in the hands of another highly qualified staff member. Little by little, over the course of two years, I had to make the administrative decision to 'retire' each of these scripts in order to make operations more efficient. Most of them were not crafted well enough to outlast my direct involvement in using them, so that I became a bottleneck in my own operations whenever they failed. The only exceptions were scripts I very heavily documented and widely distributed, at great personal effort.

Assessing Script Maturity

While we can assess the quality of scripts using traditional software quality metrics [25], *the relative importance of these metrics depends upon how script quality affects service organization maturity*. In this context, a script exhibits high quality when its use does not depend upon specialized and esoteric knowledge, so that any properly trained and authorized staff member can utilize it with predictable and helpful results. While traditional quality factors such as documentation, reliability, robustness, and maintainability remain important, the peculiar properties of the administrative environment in which we utilize these scripts suggest some new quality factors that are more relevant and focussed upon our mission:

1. *precondition awareness*: does the script understand the conditions under which it will function correctly?
 - a *detection*: can the script detect conditions under which it will not function and avoid problems?
 - b *assurance*: can the script change the system so that preconditions are satisfied?
2. *convergence*: does repeating the script produce the same effect?
 - a *self-consistency*: does repeating the script produce the same results?
 - b *non-intrusiveness*: does the script avoid repeating unnecessary intrusive actions that can potentially disrupt services?
3. *postcondition awareness*: does the script check upon what it should be accomplishing?
 - a *verification*: does the script check whether it did what it intended to do?
 - b *validation*: do script changes have appropriate external effects, as observed from another machine?
4. *atomicity*: does each script do related things as a unit, so that there are no partial effects that produce service failures?
 - a *transaction control*: is there a mechanism whereby the script can detect partial completion?
 - b *rollback*: is there a mechanism whereby the script can back out of changes made in the case of a failure?

These factors all concentrate upon assuring predictable script behavior that leaves the affected system

in a predictable and hopefully usable state, regardless of the identity of the particular administrator using the script.

Avoiding Scripting

The simplest way to avoid quality pitfalls of scripting is to utilize an automation tool whose design exhibits the above quality factors. Cfengine [4, 5, 6] and its relatives provide pre-written configuration methods possessing convergent properties. All the administrator has to do is to describe what to accomplish, and Cfengine will accomplish it in the least intrusive way. Cfengine is highly aware of required preconditions and elegantly deals with their absence. It fails predictably if it cannot accomplish its tasks. Although it does not provide transaction control, a user can craft this through careful configuration [12]. In effect, Cfengine provides most of the control one can get from a script, and assumes responsibility itself for the quality of its actions.

Cfengine is one of many tools available for avoiding scripting. My own Slink [9] solves the same problem for symbolic link tree hierarchies, and also provides a library of 'effective administrative abstractions' [10] with appropriate convergent properties for use in custom Perl scripts. Other file distribution methods that avoid or otherwise encapsulate scriptable actions include RPM (which supports scripts for custom actions) [1], rdist [8], and my own distr [11]. In my opinion, *whenever one can replace scripts with powerful, reliable, and well-documented management tools, one should*.

Unfortunately, there are many very common administrative problems that current high-quality tools do not address. While assuring appropriate contents for configuration files is relatively easy via file distribution (and interactive editing) approaches, controlling processes and other dynamic elements is a much more difficult task that usually requires some kind of custom scripting.

Short of avoiding scripts, we can better manage them and avoid writing too many. PIKT [22, 23] provides portability mechanisms for scripts that allow one script to function in a heterogeneous environment through preprocessing. Last year, we discussed how the logic programming language Prolog [12] subsumes the function of PIKT and supports script convergence, preconditions, and atomicity perhaps better than most scripting languages. But we concluded that coding in Prolog has its own unique difficulties and is not for everyone.

Then how should the mere mortals among us arrange to receive the benefits of scripting without the detriments? Providing scripts with the appropriate kinds of robustness is expensive in terms of coding labor, but utilizing naive scripts may be equally expensive in terms of administrative stability, because only people 'in the know' can deal with their deficiencies.

Interactive Scripting

To better understand the problems involved in scripting, I utilize a scripting example problem which, to my knowledge, presents almost all possible difficulties. Almost all network components have serial ‘consoles’ from which commands can be issued, and begin their lives in a state that requires some kind of manual configuration via interactive console commands. Routers, switches, and network appliances have to be assigned Internet addresses and networking information before I can utilize the Simple Network Management Protocol (SNMP) to finish the job. Typical UNIX and Linux servers must be built from the console before I can utilize automated methods to complete configuration. And if the network dies, then the only ‘sure’ method of interacting with potential culprits is still the trusty serial console.

Automating interaction with console interfaces poses many problems above and beyond just knowing how to write scripts. A human performing administrative actions must read reams of documentation, understand the meanings of commands, and adapt to messages from the console to determine future commands. A script trying to mimic these actions begins execution unaware of the device’s current state, meanings of commands, or history of changes. It must discover these by parsing dialogs as it executes.

The easy way to configure a device is through ‘invasive’ scripting that erases the whole device configuration and starts over each time. This gives the script complete initial knowledge of the state of the device by clearing all data before making changes, which in turn makes writing the script a simpler task. For example, to add a new user, one can erase all users and then create them all again, including the new ones, much to the dismay of people currently using the device!

A ‘convergent’ script [12] changes the device from an unknown initial state to a desired one, without unnecessarily interrupting concurrent use of the device. The script must discover that state, compare it with what is desired, and craft a minimal set of actions that will accomplish needed changes. This process is much more complex than simple ‘invasive’ scripting, but much more desirable because it will not interrupt the function of correctly configured devices. Most devices support this kind of interaction rather poorly. In fact,

Convergence is not a property of the device, or of its configuration, but of our ‘best practices’ in managing it while maintaining an appropriate level of service for others.

This makes crafting convergent scripts both particularly difficult and particularly important, as they embody all aspects of human interaction, including device knowledge, configuration requirements, and management policy.

My Goals

I began the work of this paper by looking for a better way to write Expect [19] or Chat scripts that will enable ‘convergent’ bootstrapping and administration of console-scriptable network nodes. I needed something like this in order to be able to reliably recover from errors made by students in building experimental networks. Left to herself, and given full reign over a network device, a student can unknowingly break SNMP (or other) management control over network devices, thus making ‘front door’ recovery techniques unreliable.

My second motivation was to bring the process of scripting closer to the ‘best practices’ I already understand. The tightest coupling I can make is to relate automated scripts to the commands I would have to issue myself in order to accomplish the same task. I consider this a much tighter ‘semantic coupling’ than, say, SNMP requests to accomplish the same changes: SNMP requests look very different indeed from the administrative commands to which they correspond.

The Lightwave ConsoleServer 3200

My example application is to create a convergent script that will automatically maintain the configuration of a LightWave ConsoleServer 3200 [30]. This is a serial console switch that allows access to any one of up to 32 serial consoles from up to 16 simultaneous incoming telnet sessions. It is remarkably easy to configure, but configuration involves setting many parameters, and these may only be set by hand using a serial command-line interface. There is no SNMP interface available and management functions are not network-accessible by any means. This device, once configured, also allows script access to all other consoles in my site, via telnet within a dedicated private (RFC1918) administrative subnet.

I wish to use the ConsoleServer in college coursework in order to give students access to remote Linux consoles, so that they may practice configuring Linux systems that are physically located in a protected location. This means that the configuration of the ConsoleServer will be changing frequently in order to allow new students access to the consoles. I already maintain databases of the students who should be given access to particular consoles. The trick is to craft a convergent mechanism by which I can assure that the appropriate students have access to appropriate machines, by adding and deleting accounts for particular students as they rotate through lab exercises.

Easy or Impossible?

One might think that this project is easy until one understands the true complexity in the interactions. Let us consider the simple subtask of making sure the switch is accessible to the correct students, and that they possess appropriate privileges. To add a user, one participates in a dialog similar to this:

```

LCI3200>login admin
PLEASE ENTER PASSWORD ****
sys admin>adduser
Number of available user records: 196
Number of users defined: 4
Enter user id | USER ID > foo
Enter case sensitive password
| PASSWORD > *****
Re-enter case sensitive password
| PASSWORD > *****
0-17 | MAX CONCURRENT LOGINS: 1 > 1
Allowed devices example:
1-5,10 | DEVICES 0 > 1-11
Allowed listen devices example:
1-5,10 | DEVICES 0 > 0
Allow user to clear device buffer
(Y/N) | YES > N
Clear screen after a command
(Y/N) | YES > Y
Enter user id | USER ID >
sys admin>logout
LCI3200>

```

(in this paper, long lines in examples are folded to fit within columns). With Chat, one can craft this specific dialog, but employing variables in the dialog is awkward at best. With Expect, one can write a TCL [24] subroutine that performs this task, where all user inputs are variables. Then one can call the subroutine multiple times to add multiple users.

This all seems relatively straightforward until we consider what can go wrong during the script. The user could have been created already, so that all we need to do is to modify settings and privileges. This requires executing a *different command* edituser. The answers to any of the above questions, as specified by our script user, could be inappropriate. In this case the device *repeats the question*:

```

Allow user to clear device buffer
(Y/N) | YES > No
Allow user to clear device buffer
(Y/N) | YES >

```

The device does not accept 'No', just 'N'. Improper answers leave the device asking the same question over and over again, so that the script must issue a control-C to reset the interface after any script failure. We must include one 'if' statement in our script to catch each possible failure.

Making the script 'convergent' requires much more work. We must teach the script how to gather data on existing users, and how to determine users that have not yet been added. Modifying users so that they have new privileges is a matter of reading each user's profile, comparing it with desired data, and changing it if necessary. Each of these processes is more complex than the example above. The net result is that the 'convergent' version of the script has an enormous number of possible execution paths, depending upon what goes wrong. This high 'branching complexity' [20] will cause the script to be very expensive to write, debug, and maintain.

If I write this script under pressure, I am obviously *not* going to have the time to do this correctly and will miss some case. So to use my script, I have to watch for failures, correct the values of parameters, and run the script again, perhaps after cleaning up after what it did the previous time. If I become impatient enough, I will modify the script to better handle some of the failures, in order of annoyance to me.

As I address the annoyances, I create another problem. As my script becomes increasingly clever, it is evolving with abandon, without functional boundaries or documentation. As it grows, it becomes likewise increasingly clever and increasingly unmaintainable. Any time the device changes, the script fails and I am the only hope of repairing it. I am well on the way to owning an irreplaceably valuable script that renders its author irreplaceable as well. This is what we call 'job security.'

It is for this reason that I used to consider the goal of creating 'convergent' interactive scripts (that apply minimally intrusive changes in order to assure device state) practically impossible.

Jackson System Design

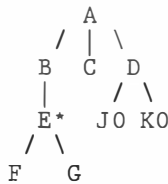
In software engineering practice, one manages project complexity and avoids this kind of developmental 'script rot' by applying a formal design methodology that controls development in order to keep scripts both understandable and reusable. Fortunately, the methodology we need was well understood during the *punched card era* of computing, and only needs to be resurrected and reapplied!

Jackson [15, 16] claimed that the way to properly design a program for processing punched card stacks is to *link the structure of the program with the structure of the stack that it processes*. He created a simplified structural model that replaces program flowcharts with 'Jackson Diagrams' that are the same for the program and its input. Each diagram depicts containment and sequence of inputs or program parts, utilizing nodes for parts and undirected edges for relationships, reading from top to bottom and left to right. The diagram:



represents a thing 'A' that consists of subparts 'B', 'C', and 'D' in that order. This thing can either be a stack of cards or the program that processes them in sequential order. Loops and branches in the program are indicated by annotating the diagram. Repeated items during a loop are starred, and optional items are annotated with a '0'.

For example, the diagram below refers to a deck of cards containing a structure A, which consists of structures B, C, and D, where B consists of multiple copies of E, and D might contain either J or K.



Jackson's key idea was to interpret this diagram also as the structure of a program to process the cards:

```

begin A;
begin B;
  for (some cases)
    begin E;
      begin F; end F;
      begin G; end G;
    end E;
  end B;
begin C; end C;
begin D;
  if (some condition)
    begin J; end J;
  if (some condition)
    begin K; end K;
  end D;
end A;
  
```

The program *must* look something like this if the cards are structured the same way.

Applying Jackson's Principle

I began this project 'expecting' to utilize an enhanced version of Expect to address state awareness and convergence problems in traditional scripts. I initially added enhanced handling of parsing and variable binding, similar to that in PIKT, in an effort to make scripts shorter and easier to understand. My approach to this problem changed dramatically in mid-project, however, when I realized that Jackson's methodology applies to the structure of the input/output streams with which we as administrators control the device. The structure of these streams (of prompts and commands) is predetermined by device design and one's intent as an operator. This pattern can be mimicked by a script in order to accomplish the same intent:

The structure of a fully functional interactive script is exactly parallel to the branching and looping structure of the device interactions in which it must engage.

This observation would have come to naught if I had used Jackson's diagrams as above, for they become unwieldy when used to describe interactions of this complexity. Fortunately, I did not need to utilize these, because one can use a variant of the Extensible Markup Language (XML) [13, 26] to perform the same function. For this, I employ the XML tags:

1. `<repeat>`: the equivalent of Jackson's star; indicates repetition of patterns within an I/O stream.
2. `<branch>`: the equivalent of Jackson's '0'; indicates that something is one of many possible options.

For example, the Jackson diagram above can be represented as:

```

<A>
  <B><E>
    <repeat><F/><G/></repeat>
  </E></B>
  <C/>
  <D>
    <junction>
      <branch><J/></branch>
      <branch></branch>
    </junction>
    <junction>
      <branch><K/></branch>
      <branch></branch>
    </junction>
  </D>
</A>
  
```

where

- `<X>` marks the beginning of X.
- `</X>` marks the end of X.
- `<X/>` marks the beginning and end of X. This is equivalent with `<X></X>`.

With this model in mind, I realized that my so-called 'high-quality scripts' looked much more like structural declarations than scripts, and that the non-structural imperative commands were obfuscating my understanding of the structure that the scripts documented. I was worrying about 'which variables to set' in the scripts, while I should have been worrying about the *structure of interactions*.

My obvious next step was to split each script into two parts: one part that documents structure and another that acts upon that structure. In the beginning, I considered the ability to separate structure from action as an extra 'toy' capability of my scripting tool. But eventually, as my structural markup language evolved in its ability to express detailed structure, I realized that:

For most purposes, crafting of individual interactive scripts can be replaced by an intelligent scripting engine that parses detailed structural specifications of the interface and its desired configuration, and then proceeds to assure that configuration by exploiting documented interface structure.

At first glance this method of scripting may seem ridiculously awkward, but in practice it is much easier, faster, and more reliable than scripting. Once the scripting engine is written, all one must do is to document the streams that it controls. This involves specifying the sequence, variant content, conditional structure, and topology of the commands that the interface understands. One accomplishes this by collecting and annotating example sessions. If one can do something manually, and react to all possible responses, one can script the process. The 'script' becomes *documentation* describing what varies in the sessions, what options or branches there are in the process, and the 'causal intent' of each interaction, e.g., creating a user.

Stream-structured Design

My variant of Jackson's method documents the structure of I/O streams through a series of simple and straightforward steps. Each step requires modifying an example script of a user session by adding XML-like markup tags. These tags form a Stream-Structure Markup Language, or SSML. When this process is completed, I feed the sum total of all sessions I have recorded, all appropriately marked and annotated, to a 'scripting engine' that I call 'Babble'. This engine utilizes structural documentation to decide how to interact with the device.

For example, let us first consider how one documents the process that deletes a user from the Lightwave 3200. The first step is to collect an example dialog that accomplishes this, using the script and cu commands upon a connected UNIX host. This produces a file:

```
Script started on Sat Sep 16 16:29:13 2000
% cu -l cua/b -b 8^M^M
Connected^G^M

LCI3200>login admin^M
PLEASE ENTER PASSWORD ****^M
sys admin>deletu^H ^Heuser foo^M
Delete user:foo Yes or No (N):Y^M
sys admin>logout^M
LCI3200>^.^M
Disconnected^G^M
% exit^M
script done on Sat Sep 16 16:30:29 2000
```

where the prefix ^ indicates invisible control characters.

Standardize Input

My second step is to remove the header and trailer, together with chaff such as ^H that indicates a backspace (together with characters I backspaced over). I convert &, <, and > to their XML equivalents &, <, and > so that they will not conflict with the XML tags I will add in the next step. I convert the remaining special characters to their Perl escape-string equivalents for easy readability and editing.

```
\r\n
LCI3200&gt;login\sadmin\r\n
PLEASE\sENTER\sPASSWORD\s****\r\n
sys\sadmin&gt;deleteuser\sfoo\r\n
Delete\suser:foo\sYes\sor\sNo\s(N):Y\r\n
sys\sadmin&gt;logout\r\n
LCI3200&gt;
```

In the above, \r represents return and \n represents line-feed, \s represents space, and \007 represents bell (control-G). After this transformation, *whitespace is ignored* in all further steps and may be used to indent for clarity.

Mark Input and Output

The next step is to annotate the remaining text so that I know what is input and what is output. There is not yet an automated way to do this, so I manually insert get and put tags to distinguish things the

interface sent from those I typed. At the end of this, untagged text will be ignored.

```
<brook name="delete">
<put>\r</put>\n
<get>LCI3200&gt;</get>
<put>login\sadmin\r</put>\n
<get>PLEASE\sENTER\sPASSWORD\s</get>
<put>****\r</put>\n
<get>sys\sadmin&gt;</get>
<put>deleteuser\sfoo\r</put>\n
<get>Delete\suser:foo\sYes
\sor\sNo\s(N):</get>
<put>Y\r</put>\n
<get>sys\sadmin&gt;</get>
<put>logout\r</put>\n
<get>LCI3200&gt;</get>
</brook>
```

I call a little part of an I/O stream a *brook* (!). One subtlety is that since the \r's are typed by us but the \n's are typed by the responding system, the \n are *outside* the respective put's. If I instead place the \n inside the put, the scripting engine will add an extra line-feed to every command, perhaps with problematic results.

Identify Variants

The next step is to document which strings vary in the stream, depending upon what I wish to accomplish with this script. I call these strings *variants* to distinguish them from traditional variables, with which they share only a superficial resemblance. In my example, only the administrative password and the name of the user to delete may vary. All else is always the same. I mark and name variants where appropriate, inserting (ignored) line breaks and indentation for readability:

```
<brook name="delete">
<put>\r</put>\n
<get>LCI3200&gt;</get>
<put>login\sadmin\r</put>\n
<get>PLEASE\sENTER\sPASSWORD\s</get>
<put>
  <var name="adminpass">****</var>\r
</put>\n
<get>sys\sadmin]&gt;</get>
<put>deleteuser\s
  <var name="username">foo</var>\r
</put>\r
<get>Delete\suser:
  <var pattern="[a-zA-Z0-9]+">foo</var>
\sYes\sor\sNo\s(N):</get>
<put>Y\r</put>\n
<get>sys\sadmin&gt;</get>
<put>logout\r</put>\n
<get>LCI3200&gt;</get>
</brook>
```

There are two kinds of variants. A *named variant* is something to be placed into put commands or discovered during get commands. There are two of these: adminpass and username. A named variant has to be assigned a value in order to be put, but *acquires* a value after a get. An *unnamed variant* is only valid within a get and represents variant input to be matched

and skipped over, documented by a Perl regular expression pattern.

In Expect, at this point, I would have to assign these values to variables outside the realm of the device language, in TCL. In SSML, these variable bindings are accomplished by comparing this description (of where variables appear) with a different XML database of appropriate values for each variant. In this way specifics of configuration are kept separate from the process by which one configures a thing.

Classify Echo Types

The next step is to carefully classify variant output into one of several classes. There are three classes of output, corresponding to echo options: normal echo (full duplex, the default), no echo, and starred echo (for passwords). These are indicated by tags that contain output within a put:

```
<put><stars>
  <var name="adminpass">****</var>
</stars>\r</put>\n
```

The default is that what I type shows up in the output in full-duplex mode. Placing output inside a stars tag documents that stars are displayed instead of what I type, while a noecho environment indicates that there is no echo at all, as in typical password dialogs.

Document Conditional Behavior

The next step is to indicate any branching or conditional behavior in the overall flow of the script. First it is possible that I will already be logged in when the script starts. In this case, I wish to skip the administrator login, a simple branch:

```
<put>\r</put>\n
<junction>
  <branch>
    <get>LCI3200</get>
    <put>login\sadmin\r</put>\n
    <get>PLEASE\sENTER\sPASSWORD\s</get>
    <put><stars>
      <var name="adminpass">****</var>
    </stars>\r</put>\n
    <get>sys\sadmin</get>
  </branch>
  <branch>
    <get>sys\sadmin</get>
  </branch>
</junction>
```

Each branch starts with a get that is used to select which branch to execute. If I receive a non-administrative prompt, I log in, else I skip the process. Appearances can be deceptive: this is *not* a method, but *documentation*. The stream can take two paths, and I have now documented both of them.

Another branch will be taken if the user name I choose does not exist. I can document this branch by collecting more data:

```
sys admin>deleteuser foo
User foo does not exist
```

To deal with this case, which is perfectly acceptable

since I wanted to delete the user anyway, I can modify the master script by adding a branch describing the new response:

```
<put>deleteuser\s
  <var name="username">foo</var>\r
</put>\n
<junction>
  <branch>
    <get>Delete\suser:
      <var pattern="[a-zA-Z0-9]+">foo</var>
      \sYes\sor\sNo\s(N):
    </get>
    <put>Y\r</put>\n
  </branch>
  <branch>
    <get>User\s
      <var pattern="[a-zA-Z0-9]+">foo</var>
      \sdoes\snot\sexist</get>
    </get>
  </branch>
</junction>
<get>
  sys admin>
</get>
```

If I receive the error message, I simply ignore it. The default action in SSML, if there is no match to a branch, is to fail with a script error.

Associating Values With Variants

The next step is to actually invoke a script engine upon the documentation in order to perform the documented function. This is a matter of binding variants to appropriate strings and calling the scripting engine to interpret the results. This in turn requires creating declarations of variants separate from – but in agreement with – the stream declarations I have made. For example, for my brook described above, I might declare:

```
<var name="adminpass">PASS</var>
<var name="username">couch</var>
```

to assign values of PASS and couch to adminpass and username, respectively. For simplicity, variant values are organized in a single global declaration in a separate file.

Repeating Commands

Most of the time, however, I do not wish to delete just one user. In SSML, I accomplish repeated tasks *implicitly* (as we did previously in Prolog [12]) by declaring sets of instances of variable values to use. I force an action to repeat by defining a variant that holds a set of instances, matched with a repeat markup that processes the instances. The structure of instances in the configuration data must be parallel to the structure of repeat tags in the markup. This process is aided by a simple but powerful name scoping mechanism.

For example, suppose that I wish to delete both users foo and bar. I create a brook to do both, by inserting the brook I have already created into a repeat context:

```
<brook name="expunge">
  <repeat instances="people">
    <insert brook="removeuser"/>
  </repeat>
</brook>
```

The variant `people` consists of two instances of data needed by `removeuser`:

```
<repeat name="people">
  <instance>
    <var name="username">foo</var>
    <var name="adminpass">PASS</var>
  </instance>
  <instance>
    <var name="username">bar</var>
    <var name="adminpass">PASS</var>
  </instance>
</repeat>
```

Each set of distinct values is called an *instance* of the process. During the `repeat`, each instance is processed in turn. During processing of a particular instance, the script engine *augments the top-level variant declarations* with new variable values for each case in turn, then invoking the brook with these new values. Variants declared outside the `repeat` clause keep their values unless shadowed by definitions inside an instance, so I could have accomplished the same effect through:

```
<var name="adminpass">PASS</var>
<repeat name="people">
  <instance>
    <var name="username">foo</var>
  </instance>
  <instance>
    <var name="username">bar</var>
  </instance>
</repeat>
```

As the variant `adminpass` occurs outside the block of instances, and is not shadowed within them, it is available to the contents of the `repeat` markup for each instance. Variant bindings during a `repeat` are *strongly typed* but *dynamically scoped*. The kind of variant (`repeat` or `text`) must exactly match its usage, but variants brought into scope by a `repeat` are available to any contained repeats or subprocess invocations. This allows one to declare multiply-dimensioned loops by defining two sets of instances with non-overlapping variant names, one set for each `repeat`.

Discovering Configuration

This variant binding scheme also works in *reverse* to allow us to inductively discover the values of variants for a set of instances. Suppose I want to get a list of all users. I know that the command for that is `listusers`:

```
sys admin>listusers
User id > COUCH
User id > FOO
User id > BAR
sys admin>
```

In my last use of `repeat`, the instances over which I iterated were all arguments to `put` commands. I can discover users by reversing this process, referring to

the variants within a `get` command:

```
<brook name="listusers">
  <put>listusers\r</put>\n
  <while instances="people">
    <get>User\sid\s>\s
    <var name="username"
      pattern="[A-Z0-9]+">
      COUCH</var>\r\n
    </get>
  </while>
  <get>sys\sadmin</get>
</brook>
```

This creates several instances, all known under the name `people`, where each one contains the username of one user of the device.

The difference between `repeat` and `while` lies in their control over instances. `repeat` does something for a *fixed* number of instances, while `while` *inductively discovers* instances as they appear in the output, and creates a list of all of them. This has the effect of *updating variant space* for the discovered values, erasing any previous value of the structured variant `people`, where each instance contains a current username, mined out of the output by using the regular expression pattern `[A-Z0-9]+`.

In this example, the `while` exits upon a timeout, after which instances discovered during each completed pass through the `while` process become instances of the `repeat` variant `people`, overwriting any previous values. At this time we know all the names of users, and could now, e.g., delete all of them by invoking the ‘`expunge`’ brook above.

Convergent Processes

Until the last example, I have not employed variants that are computed at runtime. The reader might ask what good it does to discover variable values if there is no script to utilize these values. The scripting engine itself, with appropriate guidance, can utilize this data to great advantage, or even print a new configuration file representing the *current configuration* of the device.

Assuring values of individual configuration parameters non-intrusively is fairly trivial. The engine reads them, and if they are incorrect, changes them accordingly. For it to do this, it is sufficient to instruct the engine on how to read and write particular configuration parameters, with appropriate branching to deal with different cases.

Difficulties arise, however, when one wishes to efficiently update a part of the configuration containing an unknown number of instances of a thing. For example, in assuring that my idea of current users agrees with that of the device, I start with two lists of users, one in hand and one already configured on the device. To update the users, the engine must read current user information from the device, note differences between current and desired users, and proceed to modify the configuration so that the desired information becomes current. To empower the engine to behave intelligently in this case, I must document a

few processes that act on the same variants, including how to read the user list, how to read details on one user, and how to add, delete, and modify one user's data. All of this information together describes a *convergent process* that will update the user list to have desired contents.

So far, I have used SSML to describe more or less traditional program flow that is also representable using Expect. In this example Babble transcends Expect's capabilities by *responding intelligently to documentation*. Babble reads the user list and adds, deletes, or modifies users as needed, *checking its work* at every step by reading what it has written. In this way, five short declarations are used to synthesize one incredibly complicated action that would be impractical to code as a single declaration. This complexity thus migrates from the documentation into the script engine where it belongs.

Exceptions

Some devices have particularly annoying user interfaces. For example, the 3Com Corebuilder 9000 prints the contents of SNMP traps on the console while one is trying to configure it, sometimes in the middle of typing commands. To configure this device, one must ignore these alerts while issuing configuration commands. One can do this in SSML by declaring an 'exception' pattern to check for and discard if present. This arranges for the trap data to be discarded whenever it appears, regardless of context. This would be incredibly awkward to arrange in Expect, as the exception pattern would have to be included in *every pattern match* in the whole script!

Babble

Babble is a scripting engine that parses SSML specifications and performs desired configuration tasks. It is implemented as a set of cooperating Perl packages that parse both stream documentation and variant declarations, and allow one to selectively invoke individual brooks or convergent processes with desired parameters. It is implemented as a Perl library because of the many and varied forms in which I store configuration information, in the hope that any external specification of configuration policy can be translated into an appropriate set of variant declarations using Perl. Input to each invocation consists of an I/O stream with which to interact, a compiled SSML parse tree, a name of a branch to invoke, and a multi-level associative array describing variants and sets of instances to be processed. The output of each call is the modified variant array, modified to reflect any gets executed during the script.

This version of Babble is so new that the only application I have so far crafted is the one described herein. I can report from this that crafting Babble scripts to control the 3200's configuration was accomplished unbelievably quickly, because scripts almost always worked correctly the moment they had correct

syntax according to the parser and builtin configuration tester included with Babble! Debugging was almost entirely a process of responding to complaints from Babble itself about mismatches of names, syntax errors, etc. The only specialized knowledge I needed was an understanding of how to accomplish specific things in Babble that I was used to accomplishing using scripting.

While I designed Babble specifically for serial console interaction, it can be used to automate any serial interactive process, including UNIX commands, telnet, etc, in the same manner as Expect. E.g., one can use it to parse the output of `ps` and then use the result to interact with the process table using explicit kill commands. Babble does not – and never will – have the ability to perform direct system calls. Employing solely console commands documents 'best practice' at the expense of script speed and resource efficiency, perhaps a proper decision.

Limitations

Babble of course refers to the tower of Babel from the Bible, because like the builders of the tower, it *speaks all languages, but without comprehension*. This lack of comprehension is the root of all of its permanent limitations. The commands Babble executes have no meaning to Babble itself, but are simply abstract patterns of interaction learned from experience. When that experience is somehow incomplete, it fails. When interacting with a complex device, this experience can never be complete and some failure is assured.

State Coherence

Babble's greatest weakness is nearly invisible in the example. Every script in the example requires hidden *preconditions* in order to function, and scripts must be strung together so that *the postconditions of each script satisfy the preconditions of the next*. The most important precondition is interface state, which is not even representable in the current markup language. The state of the interface indicates, e.g., whether the interface is in unprivileged or privileged mode, and whether the process is at a command prompt or within an interactive dialog. These hidden preconditions and postconditions affect the success of every script, in particular, *every subbrook of a convergent process must start from – and end within – the same exact interface state*.

Version Control

Another serious deficiency of Babble is that it cannot explicitly encode version information to assist it in dealing with identical hardware devices running different software or firmware. Each device revision requires a completely independent Babble script. This deficiency, alas, is completely intentional.

Babble's 'topological algebra' of structural tags is designed for *automated* merging of brooks that represent special cases of the same task on the same

device. This will be accomplished in the future via 'parallel tree walks' through the descriptions to be combined, in which one combined description emerges with appropriate junctions and branches inserted. So far, all attempts I have made to combine this feature with version control have compromised this automated merging capability, by making the algorithms for automated merging unnecessarily awkward or perhaps even impossible.

Branching in Babble is a *temporal* phenomenon while versioning is *spatial* in character. Problems arise in temporal merging when spatial merging has been done first; one does not have enough information to complete the merge unless the streams being merged have identical spatial structure. I consider automated merging and temporal coherence more important than version control, and believe that version control may have to be handled by a completely different tool, much as PIKT provides a metalayer for managing versions of normal shell scripts.

Paranoia

Babble's run-time checking borders on paranoia. Unlike Expect scripts, which check only for specific cues in the input stream, Babble checks for full-duplex echo of output, as well as compliance of input with all markups one specifies. Scripts abort on any deviation. Babble also frequently 'checks its work' by reading parameters it has modified.

If one wishes even *more* paranoia, Babble allows one to craft scripts that *validate* behavior rather than merely *verify*. Unlike verification, which simply checks that parameters are being set correctly, validation checks that parameters have the appropriate *external effects*. For example, after enabling telnet on a device, one can telnet into it to check that it works; after creating a user, one can login as that user from another device. These detailed sanity checks would be impractical to craft via traditional methods, but are relatively easy to craft in SSML because the engine is handling most of the details of error detection and branching.

Awkwardnesses

Alas, Babble's documentation format was driven by many expediencies. I used XML syntax because I was used to writing XML parsers. This was *not* the optimal match. Because of this, I had to escape all special characters in an awkward way so that they would not interfere with XML parsing. In fact, the syntax is *not*, strictly speaking, fully XML compliant. To make it possible to drop into Perl during a script, I had to allow embedded Perl, but in true XML one would have to escape &, <, and > in Perl code, rendering it unreadable. Thus I allow regular Perl and pre-process the documentation file, escaping all special characters in Perl scripts *before* parsing the result as XML.

This awkwardness, however, gives me the ability to incorporate features into Babble that are difficult to

code for any other base language. A Babble configuration is a 'literate program'[17] that represents several different facets of a process, including documentation, procedure, and policy, in one convenient package. The reason I chose XML for the base language of Babble was to enable me to render descriptions of these facets in HTML for viewing on any web browser. This is an invaluable debugging aid that will become a feature of Babble in the very near future. More important, *the interactions that Babble undertakes with a device can themselves be represented in XML*, so that each invocation can be described in HTML as well, with hyperlinks from the transcript of the invocation to parts of the configuration that determined its shape.

Critique

This approach is superior to writing scripts with Expect for several reasons. It avoids classical verification problems associated with script development, so that process refinement is much less dangerous than when writing real scripts. This strength is negated, however, if one employs regular programming as part of processing a stream. Babble also avoids one of the main difficulties in scripting with Expect: the need to craft complex regular expressions to parse input of irregular structure.

Avoiding Verification Problems

The greatest obstacle to using traditional scripting is that verifying the correct behavior of scripts is difficult. I avoid some of the difficulties by crafting *documentation of a pre-existing condition*, not a computer program. During the time I am tuning and perfecting documentation, the 'script engine' that utilizes the documentation remains unchanged. I thus simplify the problem of verifying my process into two problems: verifying the script engine itself as a program, and verifying the documentation as a description of external behavior. The engine need only be verified once. Verifying accuracy of its subsequent uses only requires checking its input for accuracy.

This fact makes refining a description much easier than refining a true script. The form of documentation is sufficiently simple that traditional limits to script validation do not apply. One can use automated static verification tools to exhibit possible sub-paths, validate syntax, and check correspondence between stream and parameter declaration structure. Thus one can largely avoid the problems of 'software rot' that plague the maintainers of true scripts.

The trick of separating documentation from programming only works if one can avoid embedding real program code into the documentation. If one must, one loses most of the benefits of the approach. In order to avoid this, one must be able to compile a complete parameter space before one starts the engine. If one cannot, but must compute configuration parameters 'on the fly' during the script, simple process documentation no longer suffices. Then one must drop

into Perl during a stream, so that my intended documentation now again assumes the role of a program. Of course, when one must do this, one compromises many of the strengths of the approach, in the very same way that employing embedded TCL weakens the maintainability of Expect.

Simplifying Regular Expression Syntax

While regular expression pattern matching is one of the most powerful features of Expect, TCL, and Perl, it is also one of the most dangerous and unwieldy. The parenthetic syntax for binding substrings to output variables is a source of constant confusion. The most common error is counting parentheses incorrectly so that variables are bound to incorrect values. In SSML, all parenthetic matching is *implicit* in the order of variables within a `get`, and parentheses are *not* enabled in the regular expression patterns. This makes the patterns much easier to craft and debug, and there is no danger of mismatched variables, as in parenthetic patterns or split statements.

This convention replaces one weakness with another (hopefully lesser) weakness. When crafting documentation, one must be careful to declare variants using appropriate regular expressions so that the pattern that the engine derives from your declarations is not ambiguous. For example, it is poor practice to declare two adjacent variants whose patterns create ambiguity in assigning values to the variants:

```
<var name="poor" pattern="[A-Z]+[0-9]+">
<var name="style" pattern="[0-9]*">
```

When the engine tries to match these two variants against the input 'AB1234', the first pattern match causes an early binding of the variant `poor` to 'AB1', after which the second pattern matches '234'. But matching these patterns could just as well have split the input into 'AB12' and '34', 'AB123' and '4', or 'AB1234' and ''.

Thinking Declaratively

Several limitations of SSML are entirely intentional. One cannot negate a pattern in SSML, or use a regular 'for' loop. These are not simple oversights, but based upon fundamental limits of the theory of automatic program verification.

Verifying the correctness of regular scripts without exhaustive testing is impractical. The most common method of verification is called 'weakest precondition analysis' [14, 21, 28]. To use this method, one clearly documents preconditions and postconditions of the script, and then analyzes the script line by line from end to beginning, starting from desired postconditions and carefully computing the preconditions needed to assure those postconditions. A script is 'correct' if the preconditions actually required to assure postconditions are 'weaker' (i.e., less demanding) than the stated preconditions we document. This process is easy to perform automatically for scripts containing only linear code with no loops, but is equivalent in

complexity to mathematical theorem proving for scripts containing loops in which the same variables are both set and used. Theorem proving takes far too much time to be practical.

In practice, this means that the only practical way to assure the quality of a script is to put it through a full 'regression test' after *any* change. Because of the complexity of the environment in which administrative scripts must run, this kind of testing is usually impractical or perhaps impossible. This allows incidental script bugs to remain hidden until they cause a crisis, perhaps until the original author has long ago moved on to other employment.

To be able to efficiently verify a program, one has to 'weaken' the scripting language so that limits to automatic verification do not apply. SSML documentation contains no loops that would present problems for a 'weakest precondition' verifier, unless one intentionally reads a variable and sets it inside a repeat scope *in the same brook*. Doing this in SSML constitutes a markup error that future versions of Babble will be able to detect and report.

The current implementation performs only limited verification, including reporting disagreement on names and types of variants between stream documentation and value declarations. The structure of SSML will allow future versions of Babble to locate more kinds of common programming errors, including overlap or ambiguity of regular expression patterns, as well as ambiguity of intent, such as writing data during a stream that should only be reading it, or vice versa.

Relearning Common Techniques

Writing SSML specifications requires that one learn new equivalents for common but less reliable scripting techniques. For example, it is very common for a traditional script to parse a line of input into an array with a `split` command. E.g., in Perl, one can write:

```
@parts = split(/\s+/, $line);
```

where `\s+` is a regular expression, `$line` is the unparsed line, and `@parts` is an array of fields within the line. Babble does not allow this kind of matching in a straightforward way, but there are two equivalent constructions. First, one can name all parts of the line to be matched, and match them individually:

```
<var name="first" pattern="^[^\s\n]+">
<var pattern="\s+">
<var name="second" pattern="^[^\n]+">
<var pattern="\s+">
<var name="third" pattern="^[^\s\n]+">
```

where the pattern `^[^\s\n]+` matches non-whitespace, while the pattern `\s+` matches whitespace. This will bind `first`, `second`, and `third` to the next three whitespace-delimited fields. If there are a fixed number of fields, this is the best possible documentation on their structure.

If there are several fields for which one does not know a field count, such as a list of ports separated by commas, one can instead declare a repeating structure:

```
<while instances="ports">
  <junction>
    <branch>
      <var name="port"
        pattern="[\s\n]+">
      </var>
    </branch>
    <branch>
      <var pattern=",\s+"></var>
      <var name="port"
        pattern="[\s\n]+">
      </var>
    </branch>
  </junction>
</while>
```

The complexity here is more apparent than real. This declares a sequence of input in which there are repeated instances of a port, where each pair of port numbers are separated by whitespace and a comma. The branching structure indicates that it is possible that there is whitespace in front of each instance. If, e.g., the input is '2, 5', the data that this process binds to the variant 'ports' has the structure:

```
<repeat name="ports">
  <instance>
    <var name="port">2</var>
  </instance>
  <instance>
    <var name="port">5</var>
  </instance>
</repeat>
```

This data can in turn become the argument to a repeat markup if one wishes to do the same thing to each discovered port!

Automation

Many steps in this process can be automated or streamlined so that much less user input is required. For example, when capturing example sessions, a tool that would capture and correlate both input and output (using time stamps to determine relationships) could generate the direction and echo markups that I created by hand in the example.

There are subtle semantic difficulties in automating the task any further without human intervention. For example, it is not possible to reliably infer the positions of variant data – or the regular expressions that describe them – from a few examples. A person must mark these. But once input, output, and variants are distinguished, multiple example sessions exhibiting different branches for the same task can reliably be combined automatically by parsing them, fusing their parse trees, and then printing the result. A person must nonetheless identify which set of brooks all accomplish the same task and should be fused. Likewise, after I tell the engine which scripts read and write data, the engine can automatically determine whether write operations worked or not, by reading the results and checking those against my intent.

Conclusions

When I began this work, I was possessed by the traditional spirit that scripting can solve any problem, and that all I had to do was to make scripting easier. Even when applying the relatively declarative thinking required for logic programming, I retained the old script mentality and first tried to do 'everything I could do with Perl'. This attitude was the result of 28 years of conditioning, and it took a long while to question this thinking, and even longer to unlearn old habits in order to discover ways of doing without this 'expressive power'.

The quality of our work, as script writers, is controlled by fundamental theoretical limits known to Computer Science. Normal scripts are difficult if not impossible to validate and verify by any method short of exhaustive testing. The unique properties of the administrative environment make this testing impractical, while our lack of knowledge of the complete effects of our actions hampers top-down thinking and design. Babble cannot violate any of these limits, but can carefully work around them. It discourages unproductive practices and shifts responsibility for script quality – whenever possible – away from the script itself and into a reliable intermediary component that better interfaces desires with devices.

My journey has been a 'tale of power'. Sometimes apparent power is an illusion. This illusion can cost us much time and effort to avoid. It can sap strength from our infrastructure while superficially pleasing our egos. It can keep us from realizing its effects, 'trapping us in a lifestyle' of seeming opulence with an underlying and terrible cost. But the first step in avoiding a trap is knowing of its existence.

We each seek personal empowerment in our own ways, weaving a fabric of practices and tools that gives us the stability and security we all crave as human beings. Intrinsically we all know what the real 'best practices' are: those techniques that enhance our personal empowerment and security. We may be given these by a superior, or discover them ourselves via bitter experience, but the result is the same.

If we can document these practices so that they will outlast our attention and presence, we empower others in the same way. Thus we can move beyond the 'network of trust' to form a 'network of empowerment' in which our community of administrators is much stronger than the sum of its parts. This goal requires putting the community above one's self-interest, in order that the community become strong enough to protect us better than we can protect ourselves. It requires looking beyond 'job security', toward 'mission security'. It requires acting fairly within the 'social contract' that irrevocably binds us with our organization in a pact of mutual protection and shared mission.

Because true empowerment flows not from inside ourselves, nor from our technologies, but from

caring community carefully woven around shared purpose, vision, and dreams.

Availability

Babble will be available soon in alpha release from <http://www.eecs.tufts.edu/~couch/babble>. While it is written entirely in Perl 5 and should be portable to all UNIX systems, the current version does not function properly in Linux due to a bug in the CPAN pseudo-tty module `Pty.pm` – I am working on this.

Acknowledgements

I first wish to thank intrepid system administrator Andy Davidoff for putting up with me while I learned the hard way how to be a good manager. Tufts administrators Rich Papasian, Lesley Tolman, and Tony Sulprizio were all excellent examples to me in learning this lesson. Judy Jovanelly of Lightwave Communications, Inc. was most helpful in both suggesting the Lightwave 3200 for my application, and helping me repair a trivial bug in 3200 software that Babble's engine discovered through the engine's megalomania and paranoia. Max Ben-Aaron, Robert Osborn, and Steve Moshier dedicated two lunchtimes to discussing the paper and greatly improved its content. David Krumme and Remy Evard read the manuscript and provided helpful comments. Particular thanks to my student research group, including Michael Gilfix, Noah Daniels, John Hart, and Scott Pustay, for walking alongside me on this journey of discovery, and putting up with endless discussions of what Babble can do, before it could do it.

Author Biography

Alva L. Couch was born in Winston-Salem, North Carolina where he attended the North Carolina School of the Arts as a high school major in bassoon and contrabassoon performance. He received an S.B. in Architecture from M.I.T. in 1978, after which he worked for four years as a systems analyst and administrator at Harvard Medical School. Returning to school, he received an M.S. in Mathematics from Tufts in 1987, and a Ph.D. in Mathematics from Tufts in 1988. He became a member of the faculty of Tufts Department of Computer Science in the fall of 1988, and is currently an Associate Professor of Electrical Engineering and Computer Science at Tufts. Prof. Couch is the author of several software systems for visualization and system administration, including *Seecube*(1987), *Seeplex*(1990), *Slink*(1996) and *Distr*(1997). In 1996 he also received the Leibner Award for excellence in teaching and advising from Tufts. He has assisted in maintaining the Tufts computer systems for Computer Science teaching and research since 1985, when he was a Ph.D. student. He can be reached by surface mail at the Department of Electrical Engineering and Computer Science, 161 College Avenue, Tufts University, Medford, MA 02155. He can be reached via electronic mail as

couch@eecs.tufts.edu. His work phone is +1 617-627-3674.

References

- [1] E. Bailey, *Maximum RPM*, Red Hat Press, 1997.
- [2] B. Boehm, "Software Engineering Economics," *IEEE Trans. Software Eng.* **10**, No. 1, 1984.
- [3] R. Brooks, *The Mythical Man-Month*, Addison-Wesley, Inc., 1982.
- [4] M. Burgess, "A Site Configuration Engine," *Computing Systems* **8**, 1995.
- [5] M. Burgess and R. Ralston, "Distributed Resource Administration Using Cfengine," *Software: practice and experience* **27**, 1997.
- [6] M. Burgess, "Computer Immunology," *Proc. LISA-XII*, 1998.
- [7] K. Caputo, *CMM Implementation Guide: Choreographing Software Process Improvement*, Addison-Wesley-Longman, Inc, 1998.
- [8] M. Cooper, "Overhauling Rdist for the '90's," *Proc. LISA-VI*, Usenix Assoc., 1992.
- [9] A. Couch and G. Owen, "Managing Large Software Repositories with SLINK," *Proc. SANS-95*, 1995.
- [10] A. Couch, "SLINK: Simple, Effective Filesystem Maintenance Abstractions for Community-Based Administration," *Proc. LISA-X*, Usenix Assoc., 1996.
- [11] A. Couch, "Chaos out of order: a simple, scalable file distribution facility for 'intentionally heterogeneous' networks," *Proc. LISA-XI*, Usenix Assoc., 1997.
- [12] A. Couch and M. Gilfix, "It's elementary, dear Watson: applying logic programming to convergent system management processes," *Proc. LISA-XIII*, Usenix Assoc., 1999.
- [13] C. Goldfarb and P. Prescod, *The XML Handbook, 2nd Edition*, Prentice-Hall, Inc., 2000.
- [14] C. A. R. Hoare, "An axiomatic basis for computer programming," *Comm. ACM* **12**, pp. 576-581, 1969.
- [15] M. A. Jackson, *Principles of Program Design*, Academic Press, 1975.
- [16] M. A. Jackson, *System Development*, Prentice-Hall, 1983.
- [17] D. Knuth, "Literate Programming," *Computer Journal* **27**, No. 2, 1984.
- [18] C. Kubicki, "The System Administration Maturity Model – SAMM," *Proc. LISA-VII*, Usenix Assoc., 1993.
- [19] D. Libes, *Exploring Expect*, O'Reilly and Assoc., 1994.
- [20] T. McCabe, "A software complexity measure," *IEEE Trans. Software Engineering* **2**, 1976.
- [21] B. Meyer, *Introduction to the Theory of Programming Languages*, Prentice-Hall, Inc, 1990. Chapter 9: "Axiomatic Semantics."

- [22] R. Osterlund, "PIKT: Problem Informant/Killer Tool", to appear in *Proc. LISA-XIV*, 2000.
- [23] R. Osterlund, "PIKT Web Site," <http://pikt.uchicago.edu/pikt>.
- [24] R. Ousterhout, *TCL and the TK Toolkit*, Addison-Wesley-Longman, Inc, 1994.
- [25] R. Pressman *Software Engineering: A Practitioners' Approach*, Fifth Edition, Prentice-Hall, Inc., 2000.
- [26] E. Ray with C. Maden, *Learning XML*, O'Reilly and Assoc., est. release Jan. 2001.
- [27] L. Wall, T. Christiansen, and R. Schwartz, *Programming Perl*, 2nd edition, O'Reilly and Assoc., 1996.
- [28] D. Watt, *Programming Language Syntax and Semantics*, Prentice-Hall, Inc., 1991.
- [29] The Carnegie Mellon Software Engineering Institute, *The Capability Maturity Model: Guidelines for Improving the Software Process*, Addison-Wesley-Longman Inc, 1995.
- [30] Lightwave Communications, Inc, <http://www.lightwavecom.com>.

An Improved Approach for Generating Configuration Files from a Database

Jon Finke – Rensselaer Polytechnic Institute

ABSTRACT

Much of our site configuration information is stored in a relational database, which means we need to extract this information in the appropriate format for servers and daemons. In the past we have done this with lots of little custom C programs and scripts. We have recently changed to a new approach of generating the files within the database itself using PL/SQL packages, and then using a generic file extraction program to handle the details of putting ASCII characters into Unix (or other) file systems. This has allowed us to reduce development time of programs to generate new file types, and greatly simplified supporting multiple platforms.

Introduction

At Rensselaer, we maintain a lot of our system and site configuration information in a database, using a package we developed call Simon [5, 7]. We are not alone in these efforts. I have talked with system administrators at many other sites who are working on similar projects, including the University of Alberta, Simon Fraser University, SUNY Albany, and the University of Connecticut. One of the early projects of this type was at MIT's project Athena and their Moira [15] package. This is not limited to educational sites, I have also spoken with folks at Cisco Systems and Collective Technologies about similar projects. A quick glance at the last few LISA proceedings show a number of similar projects including Accountworks [3], NFS Configuration Management [4], Unix Host Administration [16], Aurora [12], Exu [14] (Ok, they talked about doing it) and others.

In practice, a number of configuration files need to be extracted from the database. These files range from host specific files such as `/etc/printcap` [8] to platform specific files like `/etc/group` to site wide configuration files like the resource record files for the domain name system [6], to HTML and LDIF files for the University telephone directory [10]. Traditionally, each file is generated by a file specific C program, or in some cases, a SQL*PLUS script. Depending on the specific requirements of the file, the program might deal with version control (generate only when information changed), trigger post processing, and have to run under different operating systems and environments.

The general pattern when a new file type was needed was to grab one of these existing programs, change part of it and recompile it (possibly on several different versions of Unix). As the number of different versions of unix grew, and we started working with non unix platforms, maintaining all the different versions of these programs and getting them distributed

to the correct machines got to be more and more of a hassle. In addition, these programs varied widely. They used different methods of connecting to the database, supported different concepts of version control, or had additional control options, so you would at times find yourself having built on the wrong "base," and have to do even more work.

As our systems became more distributed and specialized, maintaining and developing these custom programs became more challenging, as some of the target systems did not share file systems with the development systems, tools were not available, etc. As a result, this made what should have been trivial changes in a file format (like adding a newline between records!) into an hour or more of work. For example, when we wanted to build DHCP configuration files, we wanted to build them on our DHCP servers. Unlike our existing DNS servers (Solaris) which mounted our AFS file system, the DHCP servers (OpenBSD) do not even have an AFS client available. Our existing practice of storing the file generation programs or even the configuration files in AFS space was not going to work.

Building Files in the Database

Our new approach to generating files is to generate the file in the database itself then simply write a short program to dump the "file" out of the database and into the file system of the target machine. This allows all of the file generation code to be stored and maintained in the central database, using a consistent set of tools. In addition, the program to extract the file can be generic, and once built for a particular operating environment, could be used for any of the files we might generate for that system.

Our solution to this (see Figure 1), consists of three layers. The first layer is a generic program (`Generate_File`) that runs on the target system that can connect to the database and write a file on the target system. This calls on the second layer: a PL/SQL package

[13]¹ File_Gen which runs on the database server. This module provides the sole interface to the database for the file generation program, and handles all of the access control for each of the desired files we might generate. This in turn calls the bottom layer for the file specific packages, such as Gen_RR_File, Gen_DHCP_Config and Gen_Ldif that handle the details of extracting the data and formatting it for a given type of file.

This has proven to be a very powerful technique, allowing us to keep the data extraction and formatting of the file in the database, while doing the actual file generation and version control management on our target systems. We can also use the Generate_File program for any number of different files, which means we only need to compile it once for any given platform. Adding a new file type involves writing a PL/SQL routine to format the data and registering it with the file generation package.

General Operation

To generate a file, the Generate_File program is executed with the file target as a parameter. The program connects to the database and calls the File_Gen.Get_Attr² stored procedure, passing it the file target. File_Gen.Get_Attr does some access checking, possibly calls a target specific procedure, and returns a file name, and optionally, some version information. The Generate_File program then opens the file and calls File_Gen.Get_Data routine, which calls a target specific package, and passes the result back up and the line is written to the file until a Null is returned. The file is then closed, and moved into place.

For many of the files we generate, (/etc/printcap, /etc/passwd, /etc/group, etc.), we just want a single file produced. In other cases, we want to generate more than one file at a time. For example, when we are

¹PL/SQL Packages are a collection of functions, procedures, cursors and variables. These can be both public and private. Access to the public procedures can be granted to oracle users or roles. Once accessed in a session, a package maintains state between calls.

²Procedures in packages are referenced as *package name.object name*. This is the Get_Attr procedure in the File_Gen package.

generating the resource record files for Bind, we want to regenerate all files that have changed. A single invocation of the program may create a lot of files. To handle multiple files, the Generate_File program then calls File_Gen.Get_Attr to get a new file name and the cycle is repeated until File_Gen.Get_Attr indicates that there are no more files.

Generate_File

In our current implementation, the host program is written in C, using the Oracle C pre compiler (PRO*C). However, we are not limited to using this interface. We are planning on writing one in JAVA, and one could be written in PERL or any other language that can access the database. What is more, there is no problem in mixing and matching, all interfaces can generate all file types.

In our implementation, the program does a few more housekeeping details for us. When we open a file for writing, we actually open the file *filename.new*. Everything is written there, and after it successfully closes the file, we move *filename* to *filename.old* and then move *filename.new* to *filename*. This gives us a little bit of protection from full file systems and network interruptions. It is possible for other applications to have the target file open. The script calling Generate_File program needs to be written to take the appropriate action such as signaling the other process.

The program has no special privileges. It is up to the caller to ensure that it is run in the proper directory with the proper access to the file system. In appendix 1, we have the script³ used to generate our telephone directory web pages. It is run as root via cron once a day. After setting some variables the script attempts to authenticate to AFS. The hosts.klog is a script we use to authenticate root processes on a system to AFS. Next we invoke a simple "locking" script⁴ to ensure that we are the only copy running, cd to the target directory and generate the files. The program is invoked with the "-nvr" (short for Needs Vos

³The /usr/vice/etc/pagsh is an AFSism, used to help control AFS authentication.

⁴It is crude and depends on a file existing or not. But it has not failed us in six years of operation.

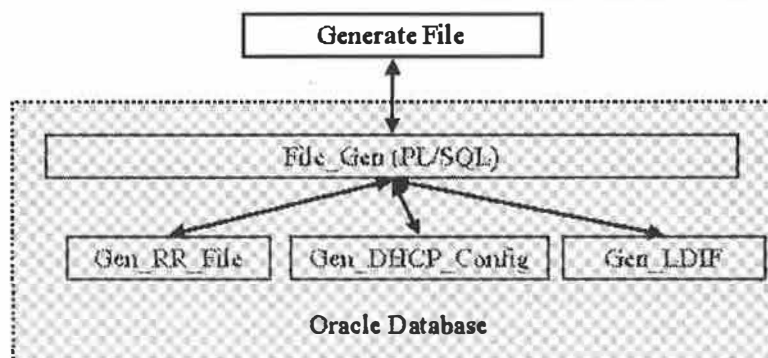


Figure 1: Service model.

Release) option. If we regenerate a file, we write that file name to the file specified by -nvr. After we are done with all the file generation, we check for the file specified by "-nvr" and if it exists, request a vos release.⁵ Instead of a vos release, other post processing could be trigger such as a "make yp," or whatever is needed.

Our version of the Generate_File program can connect directly to the database (if it is on the database machine), or can connect via SQL*NET to a remote database. In this case it can read an Oracle id and password from the command line or from a file. This works well in our environment.

File Version Control

One of the bits of information the program gets from the File_Gen.Get_Attr call is file version information. For example, we have around 170 different sub domains at our site. When we regenerate the resource record files, we don't want to build the entire set each time, but rather, we want to just update the files that have changes.

```

:
: RR file for rpi.edu
:
: Simon Database Version:26473884
:      Generation Date: 23-MAY-2000 11:07
:
:

```

Figure 2: Version Number in file.

To this end, we maintain a version number for each file. In many cases, this is written in the first 10 lines of the file with a particular flag string (see Figure 2) where we scan for the string "Simon Database Version" in the header comments of the RR file. For files that do not support comment lines, we write a parallel file with the string .vers. prepended to the file name; ie the version file for a file called passwd would be .vers.passwd.

⁵The "vos release" command is an AFSism that is used to reclone read only, replicated disk volumes.

The File_Gen.Get_Attr routine returns, along with the file name to use, an optional version number for that file and an optional flag string. If the flag string is present, it will scan for it in the target file. If there is no flag string, then it will look for the parallel version file. The program then compares the database version number with the file version, and if they don't match, the file is regenerated. The Generate_File program has a -force option to make it ignore the version numbers and generate all files.

Post Processing Control

With the version control enabled, it is possible to run the program for a given target and not write any files. In this case, no post processing would be needed. To assist in this, another option lets you specify a flag file. If a file is written, its name will be written to the flag file. Post processing code can check to see if the flag file exists and take appropriate action. We use this to automatically replicate files in AFS after generation. This is shown in appendix 1. Another example might be generating the /etc/inetd.conf file. If you updated the file, you would want to restart inetd. If there was no change to the file, there would be no need to signal the inetd process. This feature was used in some of our older programs and proved useful so we kept it as we moved to this new approach.

File_Gen Package

The core of this entire project is the File_Gen package, which acts as the gatekeeper. It provides access control and handling many of the interface details to the actual file generation procedures.

Get_Attr

When the Get_Attr procedure is called, you supply the name of the desired target (and an optional parameter; see Figure 3). It first checks to see if the current Oracle user is allowed to generate that file (access the data). If they are, it returns a file name, some control flags and some version control information. If there is some problem (unknown target, no

Name	Type		Description
Target	Varchar2	In	Name of file set to be generated.
ErrMsg	Varchar2	Out	If not null, display the message and terminate.
Filename	Varchar2	Out	Name of the file to be generated. When null, there are no more files to be generated.
DBMSOut	Varchar2	Out	If "Y", then DBMS_Output routines may be used.
Direction	Varchar2	Out	Direction of transfer, "GET" a file from the database, or "PUT" a file into the database.
Version_Number	Number	Out	The current version number of the file to be generated. If null, don't check version before generating file.
Version_Str	Varchar2	Out	If not null, scan the start of the file for this string and the version number.
Par3	Varchar2	In	A parameter passed from the command line of the caller to the file generation package. Usage defined by the individual package.

Figure 3: Get_Attr parameters.

access), the error message is returned instead. `Get_Attr` is the key routine which drives the entire generation process. Once a file is generated (or skipped due to version numbers matching), this routine should be called again for the next file.

In our initial implementation of the `File_Gen` package, we used a case statement⁶. This required changing and recompiling the package every time a new file target was added. This also made `File_Gen` dependent on all of the file-specific packages. Fortunately, Oracle provides a dynamic SQL package called `DBMS_SQL` [1] that allows a PL/SQL procedure to parse and execute an arbitrary PL/SQL routine. With these details worked out, adding a new file target was reduced to making an entry in the `Generate_File_Types` table (and writing the file-specific package of course); see Figure 4.

While the general file generation model has the `File_Gen.Get_Attr` called until there are no more files to be generated, many of our file targets only produce a single or fixed set of files. Having to write a `Get_Attr_Rtn` that keeps track of the current state (Start-File or EndFile), returns the file name, the version string, and so on was a bit of a hassle. So, if the `Generate_File_Types.Get_Attr_Rtn` is null, the `File_Gen.Get_Attr` will “fake it,” using the values supplied in the

⁶Actually, PL/SQL does not have a case statement, so it was a large `if/elsif/elsif` statement.

table. This makes it very easy to add simple file set targets. By using different sequence numbers, multiple files can be generated without having to write a file specific `Get_Attr_Rtn`. We use this to generate both our people and department LDIF files for our LDAP server.

Set_Program_Version

`Set_Program_Version` is used to make host information (current user, hostname, program version) available to the generation routines. We frequently include version information and other diagnostic info in the header of a file (see Figure 5). This is called at the start of the run, before any calls to `Get_Attr` or other routines.

Get_Data

Once a file is opened, the `Get_Data` routine is called and the result is written to the output file. If the `DBMS_OUTPUT` flag was set by `Get_Attr`, then the `Dbms_Output` buffers should also be written to the output file. This cycle repeats until `Get_Data` returns null. `DBMS_Output` [2] is a package supplied by Oracle that will buffer text provided in calls to `DBMS_Output.Put_Line` until the application retrieves the text via calls to `DBMS_Output.Get_Line`.

Put_Data

When we are “PUT”ting a file into the database, we call this routine for each line of the file. When we

Name	Type	Size	Description
Target	Varchar2	32	The name of the target file set.
Seq_Number	Number		Used to order files within a file set.
RoleName	Varchar2	255	Oracle role required to process this file.
Get_Attr_Rtn	Varchar2	65	Name of the file set specific get attribute routine.
Get_Data_Rtn	Varchar2	65	Name of the stored procedure to be called to return a line of the file.
Put_Data_Rtn	Varchar2	65	Name of the stored procedure to be called to put a line of the file into the database.
End_Data_Rtn	Varchar2	65	Name of the stored procedure to be called when all lines of the file have been read.
Def_Filename	Varchar2	65	Default file name to be used if <code>Get_Attr_Rtn</code> is null.
DBMSOUT	Varchar2	1	When “Y”, indicates that this file target may generate output via the <code>DBMS_OUTPUT</code> package.
Direction	Varchar2	4	Default direction to be used if <code>Get_Attr_Rtn</code> is null.
File_Version_Str	Varchar2	64	File version string to be used if <code>Get_Attr_Rtn</code> is null.
File_Version_Number	Varchar2	32	File version number to be used if <code>Get_Attr_Rtn</code> is null.
Package_Version	Varchar2	128	A version number (if any) for the package being used to generate the file. We manage our stored procedures with RCS and use the “\$Id\$” value here.
Package_Header	Varchar2	255	A file reference to the source file used to create this package. We use the “\$Header\$” value here.
Create_Date	Date		The date when this file target was first made available.
Update_Date	Date		The date when this file target was most recently changed.
Comments	Varchar2	255	A short description of this file target.

Figure 4: `Generate_File_Types` table.

have reached the end of the file, we call the `End_Data` routine which signals the database to do any post processing required on the data that was just loaded. Both `Put_Data` and `End_Data` have an `ErrMsg` parameter. If this is not null, the message should be displayed and the file load terminated. This allows the underlying file load packages to signal fatal exceptions.

Generate_File_Internal

There is a second PL/SQL package called `Generate_File_Internal`. From a strictly programming standpoint, this should be part of the `Generate_File` package. However, the `Generate_File` package is permitted to the public, thus all public procedures are available for anyone to use. Since the `Generate_File_Internal` defines some procedures to define and change file targets, it would not be good to let just anyone access these. Rather than put specific access control inside these routines, we kept them here and use Oracle access control to limit who can define new targets.

Add_Target_Complex

This entry point is used to add a complex file target, that is, one that provides a `Get_Attr_Rtn` to supply the file name, version info, etc. By using a procedure to add entries to the `Generate_File_Types` table, we are able to do a bit of sanity checking and maintain the

`create_date` field. If there is an existing target with the same sequence number, it is replaced with this one. You will note that the ordering of the parameters (see Figure 6 for a list) is different from the earlier procedures and table definitions. Some of the parameters are optional, so these are left to the end of the parameter list.

Add_Target_Simple

This routine is used to define simple targets. Ironically, this particular call is more complex than the previous call. Of course, since we are off-loading more of the work on the `Generate_File` package, the trade-off is ok. Like the previous routine, some of the parameters (see Figure 7) have been moved around a bit, and some of the later ones are optional.

There is also a `Add_Target_Put` which is similar to the `Add_Target_Simple`, except it defines a simple “put” entry rather than a simple “get” entry.

Target=LIST

In order to test these routines, as well as add some handy tools, this package also defines a “LIST” target that lists all the names and descriptions of all define targets. This can be handy when you need to generate a file and can’t remember the target name. (This was a very annoying lack in the previous

```
# Simon Database Version:26512881
#      Generation Date: 30-MAY-2000 18:12
#      Generation Hostname: vcmr-42.server.rpi.edu
#      Program Name:../Generate_File
#      User Name: finkej
#      Generate_File: $Id: Generate_File.pc,v 1.3
#                    2000/05/18 23:34:20 finkej Exp finkej $
#      Generate_DHCP_Files: $Id: Create_Package_Generate_DHCP_Files.sql,v
#                    1.3 2000/05/11 23:10:17 finkej$
#      Host_Dns_Maint: $Id: Create_Package_Host_DNS_Maint.sql,v
#                    1.2 2000/05/11 22:50:17 finkej Exp finkej $
#      Host_Maint: $Id: Create_Package_Host_Maint.sql,v
#                    1.2 2000/04/28 20:07:00 finkej Exp finkej $
```

Figure 5: Version information.

Name	Type	Description
Target	Varchar2	The target file set name.
RoleName	Varchar2	The role required. Note, this is checked only for the first entry for a given target.
Pack_Version	Varchar2	The version number of the package (“\$Id\$”).
Pack_Path	Varchar2	The path name of the source file of the package.
Comments	Varchar2	Comments on the package.
Get_Attr_Rtn	Varchar2	The routine name. Since this is a complex target, this must be provided.
Get_Data_Rtn	Varchar2	The get data routine name. Technically, this could be an optional parameter, but so far all complex targets have one.
Seq_Number	Number	An optional sequence number. If not specified, it will default to 10.
Put_Data_Rtn	Varchar2	(Optional) A put data routine name.
End_Data_Rtn	Varchar2	(Optional) An end data routine name.

Figure 6: Add_Target_Complex parameters

version.) There is also a version of this that lists packages and the location of the source for each of them.

File Specific Packages

Once the upper two layers are installed, the rest of the work takes place at the file specific package level. Up to this point, we did not really know nor care what files we were generating or reading. At present, we use this system to generate resource record files for BIND, host files, DHCP configuration files, /etc/passwd, /etc/group, white pages directories in ph,ldif and CSV formats and some web pages. We also use it to load in accounting records from our backup system and directory information from a remote campus. This list will be growing over time.

A rather simple example of this, is the package to generate the /etc/passwd file. A source file to create this package is available in appendix 2. Our practice is to have a source file that we feed into SQL*PLUS, the Oracle SQL interface. To facilitate test versions, we use a "define name=" statement to set the package name. This is substituted in the appropriate places in the file when executed in SQL*PLUS. During development, we can change the name and not risk wiping out the production version.

The source file has three sections: defining the package specification; the package body and finally, registering the new target with the Generate_File routines.

The prompt statement prints the message. This proves helpful when there are errors, as you get a better idea which section had the failure. The first section, defining the package specification (Create or Replace Package &name) is pretty simple. It includes an RCS "\$Header\$" and the definition of the Get_Data routine. The parameters P1 and P2 are optional parameters that are passed in from the command line. Their usage is based on specific package. They might be used to enable debugging or in some way control details of the file generation.

The second section, defining the package body, (Create or Replace Package Body &name) has the actual PL/SQL code that is executed to produce the password file. The cursor sets up the query to extract the data. This happens on the first call to Get_Data when the cursor is opened. Each fetch statement brings the next row of data into the "R" variable. This record is automatically defined based on the columns in the select statement. The quoted strings in the select statement are column aliases. Finally, when the end of the data is reached, a null is returned rather than a PW file entry and the cursor is closed.

The third section is used to register this file type with the Generate_File routines. Since this is a simple case, we can use the Add_Target_Simple routine, define the target name, skip access checks, include the version info and some comments and set the output file name to be passwd_demo. Finally, we provide the name of the get data routine. We do not specify a sequence number, so this will take the default of 10, and replace any existing definition for etc-passwd_demo.

Conclusions and Futures

Execution Time

In some casual timing tests we found mixed results. For a simple extraction, where all data is in a single table, the custom program appears to be slightly faster than the PL/SQL approach. However, if the file is complex, requiring data from many tables, the PL/SQL approach is faster. This is in part because more of the work taking place on the database server, thus reducing the I/O between the server to the application.

There is also some overhead from using the Dynamic SQL codes rather than hard coding the switch statement. Some sample runs show about a 15% increase in run time by using the dynamic sql routines. However, given the reduction in development time, this seems worth the tradeoff. If this

Name	Type	Description
Target	Varchar2	The target file set name.
RoleName	Varchar2	The role required. Note, this is checked only for the first entry for a given target.
Pack_Version	Varchar2	The version number of the package ("Id\$").
Pack_Path	Varchar2	The path name of the source file of the package.
Comments	Varchar2	Comments on the package.
Filename	Varchar2	The name of the file we will generate.
Get_Data_Rtn	Varchar2	The get data routine name.
Seq_Number	Number	An optional sequence number. If not specified, it will default to 10.
DbmsOut	Varchar2	The value for the DBMS Output flag. Should be "Y" or "N".
File_Vstr	Varchar2	An optional version string to be used with version control.
File_Vnum	Varchar2	An optional version number.

Figure 7: Add_Target_Simple

difference is a problem, it would be possible to write a program to generate the source file.

Development and Deployment Time

The development time to add a new file to be generated to the system has been significantly reduced, in part because all of the changes are now isolated to the central database, rather than being distributed across many remote servers. Once the `Generate_File` program is installed on a machine, virtually no work needs to be done on that machine.

The PL/SQL language is well integrated in the Oracle environment, and many of my PL/SQL programs are noticeably shorter than the PRO*C programs they replaced. For example, the original PRO*C version of the program to generate the faculty/staff phone directory was 2639 lines of C/PRO*C code versus the 1588 lines of PL/SQL that replaced it. I feel that the PL/SQL was faster to write as well as being shorter.

Another factor that reduced development time, is that we often are doing a web component for the front end of these files. This results in a set of PL/SQL routines to access the tables, so half the work is done by the time we need to write a file generation routine.

Another advantage is that we are able to use display code used by the Oracle web server when we are generating static HTML pages. We are using this convergence to dynamically preview static web pages which reduces the development time with those as well.

Handy Tricks

To help validate post processing, when we generate our RR files for bind, we include an entry at the end that includes a "wc"⁷ for the entire file in one of the fields. This allows post processing scripts to quickly determine if they are working with a complete passwd file, and not one that got truncated. Adding support for this to the system could be handy.

For our generated password files, we include a special entry at the end (show here folded):

```
TIMESTAMP:##TIMESTAMP:397:4000:
13834 37930 1236688
2000-09-20-13-01-07:/tmp:/bin/true
```

The GECOS field has a "wc" value, as well as the time and date of generation. This is handy when your password file distribution processes hit snags.

New Platforms

We are in the process of developing a JAVA version of this program to run on Linux systems. We don't have the Oracle development tools licensed for this platform, and our network servers are moving away from our centrally administrated AIX machines to this platform. This is actually being driven by our security team wanting to process the DHCP lease file⁸.

⁷Unix wordcount; returns lines, words and bytes in a file.

⁸It is somewhat ironic that this paper started with describing how we generate DHCP configuration files and has come back around to the other side of that project.

New Directions

Since we already support "GET" and "PUT" operations, what else could we want? Well, rather than getting data from a file, how about getting data from a program. We currently have a custom C program that runs `lscfg` on a pipe, captures the output, digests it a bit and saves it into the database. We could replace it with a file set that does the same thing. What is more, if we want to add additional program runs, we can do that by changing the package stored in the central database and we no longer have to worry about getting new versions of the custom programs out to our clients. I expect that we will have our client machines running `Generate_File` with the file target of "Self Exam" periodically to bring information on the config back to our central management system[11]. One limitation of this is that it only works with text files. Trying to pick up binary files such as `/var/adm/wtmp` might still require custom programs such as the one we wrote to assist in doing demographic analysis of workstation use [9].

Along with running programs and capturing their output, we could run the other way and generate text and pass it to a program. This could be used as a quick and dirty interface to `lpr` or mail. One of the parameters available to the file generation packages is the platform (Hardware, OS) so it could presumably make reasonable assumptions as to what programs it can call, and where to find them. This support could also be used to initiate post processing, although that might better be handled in the calling script.

Internal Changes

I expect that we will merge the `Generate_File_Internal` package back in to the `Generate_File` package and build the access checking into the add target routines.

References and Availability

All source code for the Simon system is available on the web. See <http://www.rpi.edu/campus/rpi/simon/README.simon> for details. In addition, all of the Oracle table definitions as well as PL/SQL package source are available at <http://www.rpi.edu/campus/rpi/simon/misc/Tables/simon.Index.html>.

While the specific file generation packages are Rensselaer specific, both the host programs and the `File_Generate` should be pretty generic and could be used at other sites with no modifications.

Acknowledgments

I would like to thank Alan Powell and Jackie Stampalia of Server Support Services, Rensselaer Polytechnic Institute for their willingness to read and comment on many drafts of this paper. Special thanks go out to Remy Evard of the Argonne National Laboratory who helped edit the final version of the paper.

Author Information

Jon Finke graduated from Rensselaer in 1983, where he had provided microcomputer support and

communications programming, with a BS-ECSE. He continued as a full time staff member in the computer center. From PC communications, he moved into mainframe communications and networking, and then on to Unix support, including a stint in the Nysernet Network Information Center. A charter member of the Workstation Support Group he took over printing development and support and later inherited the Simon project, which has been his primary focus for the past nine years. He is currently a Senior Systems Programmer in the Server Support Services department at Rensselaer, where he continues integrating Simon with the rest of the Institute information systems. When not playing with computers, you can often find him building or renovating houses for Habitat for Humanity, as well as his own home. Reach him via USMail at RPI; VCC 319; 110 8th St; Troy, NY 12180-3590. Reach him electronically at finkej@rpi.edu. Find out more via <http://www.rpi.edu/~finkej>.

References

- [1] Eric Armstrong, Steve Bobrowski, John Frazzini, Brian Linden, and Maria Pratt, *Oracle 7 Server Application Developer's Guide*, chapter 11, pages 1-22. Oracle Corporation, Dec 1992.
- [2] Eric Armstrong, Steve Bobrowski, John Frazzini, Brian Linden, and Maria Pratt, *Oracle 7 Server Application Developer's Guide*, chapter 6, pages 23-28. Oracle Corporation, Dec 1992.
- [3] Bob Arnold, "Accountworks: User Create Account on SQL, Notes, NT and Unix," *The Twelfth Systems Administration Conference (LISA 98) Proceedings*, pages 49-61, Sybase Inc, USENIX, December 1998, Boston, MA.
- [4] Fabio Q. B. da Silva, Juliana Silva da Cunha, Danielle M. Franklin, Luciana S. Varejao, and Rosalie Belian, "An nfs configuration management system and its underlying object-oriented model," *The Twelfth Systems Administration Conference (LISA 98) Proceedings*, pages 121-130, Federal University of Pernambuco, USENIX, December 1998, Boston, MA.
- [5] Jon Finke, "Automated userid management," *Proceedings of Community Workshop '92*, Troy, NY, June 1992, Paper 3-5.
- [6] Jon Finke, "Simon system management: Hostmaster and beyond," *Proceedings of Community Workshop '92*, Troy, NY, June 1992, Paper 3-7.
- [7] Jon Finke, "Relational Database + Automated Sysadmin = Simon," Invited Talk, July 1993, Sun Users Group - East Conference, Boston, MA.
- [8] Jon Finke, "Automating printing configuration," *USENIX Systems Administration (LISA VIII) Conference Proceedings*, pages 175-184, USENIX, September 1994, San Diego, CA.
- [9] Jon Finke, "Monitoring Usage of Workstations With a Relational Database," In *USENIX Systems Administration (LISA VIII) Conference Proceedings*, pages 149-158, USENIX, September 1994, San Diego, CA.
- [10] Jon Finke, "Institute White Pages as a System Administration Problem," *The Tenth Systems Administration Conference (LISA 96) Proceedings*, pages 233-240, USENIX, October 1996, Chicago, IL.
- [11] Jon Finke, "Automation of site configuration management," *The Eleventh Systems Administration Conference (LISA 97) Proceedings*, USENIX, October 1997, San Diego, CA.
- [12] Xev Gittler, W. Phillip Moore, and J. Rambhasker, "Morgan Stanley's Aurora System: Designing a Next Generation Global Production Unix Environment," *Ninth Systems Administration Conference (LISA 95) Proceedings*, USENIX, September 1995, Monterey, CA.
- [13] Tom Portfolio, *PL/SQL Release 8 User's Guide and Reference*, Oracle Corporation, December 1997, Part No. A58236-01.
- [14] Karl Ramm and Michael Grubb, "Exu - a system for secure delegation of authority on an insecure network," *Ninth Systems Administration Conference (LISA 95) Proceedings*, pages 89-93, Duke University, USENIX, September 1995, Monterey, CA.
- [15] Mark A. Rosenstein, Daniel E. Geer, Jr., and Peter J. Levine, "The Athena Service Management System," *USENIX Conference Proceedings*, pages 203-211, MIT Project Athena, USENIX, Winter 1988.
- [16] Gregory S. Thomas, James O. Schroeder, Merrill E. Orcutt, Desiree C. Johnson, Jeffrey T. Simmelink, and John P. Moore, "Unix Host Administration in a Heterogeneous Distributed Computing Environment," *The Tenth Systems Administration Conference (LISA 96) Proceedings*, pages 43-50, Pacific Northwest National Laboratory, USENIX, October 1996, Chicago, IL.

Appendix 1: DirectoryGen.sh

```
#!/usr/vice/etc/pagsh
# Script to update Directory files from Simon
HTMLROOT=/afs/.rpi.edu/dept/acs/rpinfo/common/AutoGen
DirPgmRoot=/campus/rpi/simon/directory/2.0/@sys/bin
HTMLROOT=/afs/.rpi.edu/dept/acs/rpinfo/common/AutoGen
LOCKPGM=$ROOT/common/bin/wait_for_lockfile.sh
LOCKFIL=$ROOT/Dirgen_Sync_Lock
VOSRELEASE=$ROOT/common/bin/SQLVosRelease.sh
HNVR=$HTMLROOT/Needs_Vos_Release
# Define Oracle variables
LOGNAME=SimonXfr ; export LOGNAME
ORACLE_HOME=/opt/app/oracle/product/8.0.5 ; export ORACLE_HOME
ORACLE_SID=SIM3 ; export ORACLE_SID
PATH=$ORACLE_HOME/bin:$PATH ; export PATH
# Get a token
if [ -x /usr/local/etc/host.klog ] ; then
    /usr/local/etc/host.klog -t
else
    echo "Unable to get token"
    exit 1
fi
# Get a lock
if ( $LOCKPGM $LOCKFIL 10 ) ;
then
    echo "Unable to Lock $LOCKFIL"
    exit 1
fi
# Regenerate directory HTML files
cd $HTMLROOT
dirdeptgen=$DirPgmRoot/Generate_File
dirdeptpar="-target department_html -nvr $HNVR"
if [ -x $dirdeptgen ]; then
    su $LOGNAME "-c $dirdeptgen $dirdeptpar"
else
    echo "Unable to run $dirdeptgen"
    exit 1
fi
# See if we need a vos release
if [ -f $HNVR ];
then
    echo "==>[ignored: n]<== Vos releasing for"
    cat $HNVR
    if [ -x $VOSRELEASE ];
    then
        $VOSRELEASE simongen
        rm $HNVR
    else
        echo "Unable to locate sysctl, vos release aborted"
    fi
fi
```

Appendix 2: Generate_Passwd Package

```
define name=GENERATE_ETC_PASSWD_DEMO
prompt Creating package &NAME
create or replace package &name as
-- $Header: /afs/.rpi.edu/campus/rpi/simon/prop/2.0/init_sql/RCS/
Create_Package_Generate_Etc_Passwd_Demo.sql,v 1.1 2000/09/20
01:11:41 finkej Exp finkej $
-- Generate a /etc/passwd file from the logins table
```

```

procedure get_data(result out varchar2, p1 in varchar2, p2 in varchar2);
end &NAME;
/
prompt Creating package body &NAME
create or replace package body &name as
Cursor Get_Pw_Ent_Curs is
Select Username "UNAME", Unixuid "UID", nvl(Unixgid,4000) "GID",
       Public_Personal_Info "GECOS"
  from Logins
 where source like 'PRIMARY%'
    and when_marked_for_delete is null
 order by unixuid;
procedure get_data(result out varchar2, p1 in varchar2, p2 in varchar2)
is
  R          Get_pw_Ent_curs%Rowtype;          -- The data from Oracle
begin
  --
  -- First time through, we open the cursor
  if not Get_Pw_Ent_Curs%IsOpen
  then
    Open Get_Pw_Ent_Curs;
  end if;
  --
  -- Get the data into R, a record based on the query
  Fetch Get_Pw_Ent_Curs
    into R;
  if Get_Pw_Ent_Curs%NotFound
  then
    -- End of data, close the cursor and return a null
    Result := Null;
    Close Get_Pw_Ent_Curs;
  else
    -- Build the password file entry
    Result := R.Uname || ':' || to_char(R.Uid) || ':'
              || R.Gid || ':' || R.Gecos || ':'
              || '/home/' || ltrim(to_char(R.uid mod 100, '09')) || '/' || R.uname
              || ':/bin/bash';
  end if;
end get_data;
end &name;
/

-- Call Add_Target_Simple to register this file target.
begin
Generate_File_Internal.Add_Target_Simple(
  'etcpasswd_demo',          -- Target Name
  null,                      -- Let ANYONE do this
  '$Id: Create_Package_Generate_Etc_Passwd_Demo.sql,v 1.1 2000/09/20
                                01:11:41 finkej Exp finkej $',
  '$Source: /afs/.rpi.edu/campus/rpi/simon/prop/2.0/init_sql/RCS/
                                Create_Package_Generate_Etc_Passwd_Demo.sql,v $',
  'Generate a /etc/passwd file as a demo for LISA',
  'passwd_demo',            -- Output File Name
  '&name..Get_Data');       -- Routine Name defined above
end;

```

Fokstra and Samba – Dealing with Authentication and Performance Issues On A Large Scale Samba Service

Robert Beck & Steve Holstead – University of Alberta

ABSTRACT

At the University of Alberta, we have approximately 55,000 user id's using central services authenticated by Kerberos. We use AFS for central file service. We use Samba to provide Windows compatible access to much of our central file service. Samba contains a number of useful features for Microsoft Windows compatibility, including a kludge to deal with the problem of Windows sending an all uppercase version of a user's password. We observed that when Windows connects to a share, it frequently attempts many incorrect passwords repeatedly before trying the correct one. This created a very heavy authentication load on our central Samba service when users would connect every morning and authenticate. We observed this load and noticed that most of our problems were caused by repeated attempts to authenticate, and the high cost of checking these attempts.

To help reduce the load due to authentication, we implemented FOKSTRAUT, a set of modifications to Samba to cache recent password failures and successes in a DBM database built by the Samba server as it runs. By caching the recent failures we avoid expensive re-checks of the (many) other passwords Windows likes to send us. We also cache the correct case of the real password, and by doing so we avoid the expensive overhead of "cracking" an all uppercase password When Windows decides to send one. We also use FOKSTRAUT to cache the NT and LanMan password hashes of a users password once we see a successful authentication. This then allows us to use the newer Windows NT password hash after the user has connected once, without having to centrally convert and maintain a large SMB password file, and while maintaining the ability of our server to access services such as AFS which can not be authenticated against using the Windows password hash alone. Performance on our service has been drastically improved since the implementation of FOKSTRAUT.

Introduction

Our experience started with a performance problem. We are a large site which uses Kerberos [1, 2] with approximately 55,000 user ID's. We have a large scale Samba [4] service which serves up files from our central AFS [3] file service using the Microsoft Windows (Windows) [9] SMB protocols [5, 6]. It was and is exceedingly popular with our on-campus clients who use it to access centrally maintained file space when they must use a Windows machine. The problem we had was that while the service would perform well during the day, it would fall over under a crippling load in the mornings when users would establish their initial connections. This problem hit us rather suddenly this last year when various factors resulted in a large increase in use at the start of a school term. We then set out to determine the source of the problem and what, if anything, we could do about it.

The Source of our Problem

We examined the server at various times, and we found that it was completely CPU bound in the mornings when users were connecting, with many individual smbd processes competing for CPU. Once this had

been determined, we set out to find out where it was spending its time by adding a few key little diagnostic printf's to our Samba code. Watching the users connect in the morning we found the source of our CPU hogs.

On connection, many of our users' machines would try several times with a password that obviously wasn't their password on our service. Several attempts had to be made, and fail, before the correct password would be given by the Windows client machine. The result of this was that our server spent a lot of time failing the same password, for the same user, over and over again.

Our Kerberos passwords may only be changed through a password changing mechanism that forces the use of "good" passwords, and does dictionary and pattern checks against any attempts. Very frequently on our server, users would attempt to connect initially with very bad, easy-to-guess passwords, and then only after several failures, would try the correct password. These initial passwords which the connection would attempt and fail were the same as the users "Windows" password (NT domain or otherwise) with alarming regularity. So, by configuration or design,

many users' Windows machines were determined to tell us their local passwords before attempting to give us the one we wanted to authenticate them with Kerberos. This has interesting security implications – it's easy for our clients to configure Windows to leak their passwords to an SMB server. For the purposes of this exercise, we were only concerned about the performance implications.

In addition to the repeated attempts of an incorrect password, many of our users' Windows machines will attempt to use an all uppercase password. We have little or no control over the client software version, and so were not able to avoid this problem at the client end. This means we have to support the password level [10] "crack" feature in Samba where the server would repeatedly try different case combinations. While we knew this was a possible cause of the heavy load generated by authentication, the effect was greatly magnified by the repeated attempts of incorrect passwords – each incorrect password had to be checked repeatedly in multiple cases.

We were faced with a huge overhead on authentication which involved "cracking" passwords, and doing it repeatedly on the same bad password. Short of abandoning our Windows clients we had to find a way to deal with this on our Samba service.

The NT Password Hash

Samba itself supports the Windows NT encrypted password scheme. To use this, a UNIX administrator creates an "smbpasswd" file [11] in which the Windows NT hash of the user's passwords are stored. In this method, the client machine passes the Windows NT style hash, and is not faced with the problem of case insensitive password. One possibility we examined was that of converting our service to require this scheme. We could not do this for two reasons:

1. We maintain our central ID's in Kerberos, We did not want the added administrative cost of maintaining a 55,000 user flat file of SMB passwords.
2. We access AFS file space from this server – There would be no way for the server to get an AFS token to access the user's file with only the NT password hash (and not the user's Kerberos password or a ticket). This will be a

problem with any external service that can not be authenticated against using the NT password hash.

Our Solution: FOKSTRAUT

Having ruled out doing it the "correct" way for Windows by using the NT password hash, we tried to examine what would give us a solution for our environment. First, we made a few observations:

1. The users that handed us passwords that failed would usually try them twice before proceeding to the next password (sometimes another bogus one, and sometimes the right one). This was what most of our CPU bound processes were doing – busily cracking a bogus password.
2. If we knew the real password, and were given a case-insensitive version of it, we could start by checking the "correct" case and avoid the expense of repeatedly trying to crack the password.
3. If we knew the real password we could also compute the NT password hash.

Knowing this, we implemented FOKSTRAUT. FOKSTRAUT is a password success and failure cache for Samba. It uses a DBM database indexed by the user name to store the most recent passwords that failed, as well as the last password that worked for each user. It stores the successful passwords both as the clear text password, and as it's Windows NT hash, enabling the FOKSTRAUT cache to be used to authenticate Windows clients passing a Windows NT password hash for a user that has been seen before. Since it stores the clear text password along with the NT password hash, the server then knows the clear text password of a user authenticated with the NT password hash and can use this to authenticate the user for other services (AFS in our case).

How FOKSTRAUT Works

When our Samba server starts up, it checks for the FOKSTRAUT database, a DBM database indexed by user name. This database is created if it does not exist. The Samba server then stores and retrieves from this database each time a user authenticates.

FOKSTRAUT keeps track of the following information:

- The three most recent failed passwords, and a count of how many times each password has

```
Sep 28 07:23:41 samba smbd[76722]: trying password AUCTIONS for user luser
Sep 28 07:24:08 samba smbd[76722]: Bogus password, user luser, password AUCTIONS
Sep 28 07:24:11 samba smbd[76722]: trying password AUCTIONS for user luser
Sep 28 07:24:47 samba smbd[76722]: Bogus password, user luser, password AUCTIONS
Sep 28 07:24:49 samba smbd[76722]: trying password AUCTIONS for user luser
Sep 28 07:25:18 samba smbd[76722]: Bogus password, user luser, password AUCTIONS
Sep 28 07:25:20 samba smbd[76722]: trying password Ac94metoo for user luser
Sep 28 07:25:20 samba smbd[76722]: worked first time for user luser, password Ac94metoo
```

Figure 1: Example of our diagnostic log output. Note the times between success and failure: during this time this smbd was busy using CPU attempting combinations of this password.

been given and failed since the last successful connection.

- The last successful password, stored as clear text, NT Hash, and LanMan hash formats.
- An “smbpasswd” entry [11] for the user – this is used so that we can take advantage of Samba’s NT and LanMan password features if we find a user in the database.

When a user connects to our FOKSTRAUT modified Samba server, the server will look up and retrieve the information about previous connection attempts in the FOKSTRAUT database. It then compares the password the user is attempting to any previously cached failed passwords. If that password is the same as a previously cached failure, the connection is immediately failed without checking against the system, and the failure count for that password is incremented in the database, and saved for the next time. When the failure count for a failed password reaches three, this password is removed from the database. This ensures that a cached password failure can not continuously fail without being re-checked against the system authentication methods.

Assuming the password didn’t match one of the cached failures, the password is then case insensitively compared to the password recorded as being successful previously. If the passwords match, the correct case stored in the database is used, and authentication proceeds against the system authentication methods. If authentication is then successful, this connection succeeds. All previous failure counts are cleared back to zero, and the resulting record is stored back into the database for the next time.

If the password presented doesn’t match anything FOKSTRAUT knows about we attempt to authenticate in the usual manner. If the password succeeds, the correct case is recorded in the cache, the NT and LanMan hashes are computed, and the entry is saved as in the above case. If the new password fails, it will be added to the cache of failed passwords, replacing the least used (least failed against) of the three saved entries.

If our modified Samba service receives an NT or LanMan hashed password, it checks it against the entry saved in the FOKSTRAUT database. If it is present, an smbpasswd entry from the database is used for the user, and authentication proceeds as it would in the usual case of an smbpasswd based connection in Samba [4]. If that authentication then succeeds, in our case, the server then gets Kerberos Tickets and an AFS token for the user based on the saved clear text password from FOKSTRAUT. At this point if both the NT hash authentication and the Kerberos/AFS authentication are successful, the connection is deemed to be successful. Otherwise, it is considered a failure, and the cache entries (including the password hashes) are cleared.

The choice we made of saving three different cached failures was based on our own observations of

what we saw from our users. Three different bogus passwords failing was the most we typically saw before seeing the real password. The choice of failing against a cached failure up to three times before we checked again against the system was based both on our observations of the typical number of tries we would see, along with a desire to make sure we didn’t hold up a user unduly in the (unlikely) case where they make a connection attempt, the password fails, and they then change their password to what they just recently failed with.

Advantages

We have seen two primary results:

1. We have seen a huge performance improvement, we can now easily deal with the several hundred users we get every morning where we could not before. Simple load average during peak times has dropped from peaking at 30 to peaking at 3, with a sustained average dropping from over 10 to less than 2.
2.) We now have a mechanism by which we can support the non-cleartext password SMB connections to our server, while still making use of Samba as a gateway to AFS, Users are simply told to connect once from a login server, or with a Windows machine set to send cleartext passwords on connection, and then after that the non-cleartext SMB connection will work.

We are very happy with the increased level of performance these modifications have given us.

Date		runq-sz	%runocc
Tue Feb 29 08:30:05 2000		3.700	80.000
Tue Feb 29 08:40:05 2000		8.900	100.000
Tue Feb 29 08:50:05 2000		10.800	100.000
Tue Feb 29 09:00:06 2000		12.600	98.000
Tue Feb 29 09:10:06 2000		17.500	100.000
Tue Feb 29 09:20:14 2000		21.400	97.000
Tue Feb 29 09:30:51 2000		4.500	97.000

Figure 2: Run queue size and utilization, 8:30 AM to 9:30 AM, week before implementation.

Date		runq-sz	%runocc
Tue Mar 7 08:30:06 2000		1.700	37.000
Tue Mar 7 08:40:06 2000		1.400	48.000
Tue Mar 7 08:50:06 2000		1.800	55.000
Tue Mar 7 09:00:06 2000		1.800	60.000
Tue Mar 7 09:10:06 2000		1.400	37.000
Tue Mar 7 09:20:06 2000		2.000	53.000
Tue Mar 7 09:30:06 2000		1.800	28.000

Figure 3: Run queue size and utilization, 8:30 AM to 9:30 AM, week after implementation.

Disadvantages

FOKSTRAUT has several disadvantages. The first one that comes to first one that comes to mind is security. If a Samba server is compromised and is running FOKSTRAUT, all the users’ passwords are available to the attacker through the database. This is a

serious concern to us, and we therefore ensure that FOKSTRAUT is only run on a dedicated, secured machine running no other services and that there is no access by regular users to the machine. We judge this risk as acceptable to us, given that even if the machine were not running FOKSTRAUT, an attacker compromising the Samba server could install a trojaned daemon and collect the same passwords with very little extra effort. The results of a compromise of the Samba server would be bad for us with or without FOKSTRAUT, considering that we must run Samba with cleartext passwords enabled.

The other drawback from the point of view of the Windows NT passwords is that the server does not know the Windows NT hash until the client has successfully authenticated to it once so that the server knows the cleartext password. We get around this by providing a simple script on a login server in which a user can make an SMB connect and authenticate themselves, making themselves “known” to the Samba server. Alternatively, the users can make an initial connection to the service from a Windows machine set to send the clear text password, then switch to using the NT password hash.

Conclusions

A password cache is a major performance win when used with Samba service that must support clear text passwords and can not rely exclusively on the NT password hash protocol. In our case, with the ability to use the NT password hash for authentication and still have the server know the real password, it enables us to use the NT password hash connection mechanism while still supporting our external authentication mechanism (AFS) which is incompatible with the NT password hash. The performance wins we saw changed our service from one that would collapse under the load to one that now remains up for months at a time, providing reliable file service. If you are in a position where you have to offer large scale Samba service, we think this is definitely something to consider.

Availability

Our code is available as a patch for a current Samba distribution. It has been tested and used on OpenBSD [7] and AIX [8] (and should port very easily to anything else remotely Unix-like). It is made available under BSD style license terms, and can be obtained at <ftp://sunsite.ualberta.ca/pub/Local/People/beck/fokstraut/>.

Author Information

Bob Beck has a Masters degree in Computing Science from the University of Alberta. He has worked in a variety of systems administration and programming positions at the University of Alberta since 1990. He also works as a consultant, instructor, and

programmer with Obtuse Systems Corporation. He is currently the Secure Systems Specialist for the University of Alberta, as well as working on several free software projects, especially OpenBSD. You can reach him by postal mail at Computing and Network Services; 352 General Services Building, University of Alberta Campus, Edmonton, Alberta, Canada, T6G 2H1. You can reach him by e-mail to beck@bofh.ucs.ualberta.ca or beck@obtuse.com.

Steve Holstead has been responsible for programming and system administration duties working at the University of Alberta since 1978. His experience has taken him from the centralized mainframe systems support to a distributed client server environment in which he works today. He can be reached via e-mail to Steve.Holstead@ualberta.ca.

References

- [1] J. Steiner, C. Neuman, and J. Schiller, *Kerberos: An Authentication Service for Open Network Systems*, Usenix Association's Conference Proceedings, Dallas, Texas, February, 1988, p. 191-202.
- [2] J. Kohl, and C. Neuman, *The Kerberos Network Authentication Service (V5)*, September 1993.
- [3] *AFS filesystem information from Transarc Corporation*, <http://www.transarc.com/Product/EFS/AFS/index.html>.
- [4] *The Samba project*, <http://www.samba.org/>.
- [5] *CIFS information*, <http://anu.samba.org/cifs/>.
- [6] Sharpe Richard: *Just What is SMB? (v 1.2)*, <http://anu.samba.org/cifs/docs/what-is-smb.html>.
- [7] *The OpenBSD Project*, <http://www.openbsd.org/>.
- [8] *The IBM AIX Operating System*, <http://www.ibm.com/servers/aix/>.
- [9] *Microsoft Windows*, <http://www.microsoft.com/>.
- [10] “Password Level,” *Samba Documentation*, <http://us1.samba.org/samba/ftp/docs/htmldocs/smb.conf.5.html>.
- [11] “smbpasswd,” *Samba Documentation*, <http://us1.samba.org/samba/ftp/docs/htmldocs/smbpasswd.5.html>.

Designing a Data Center Instrumentation System

Bob Drzyzgula – Federal Reserve Board

ABSTRACT

This paper describes the author's efforts in designing an external, out-of-band hardware monitoring and control system for use with microcomputer-based server, storage and communications systems deployed in a data center environment. This system, when complete, will consist of a collection of microcontroller-based monitoring nodes, one per monitored device. Each of these intelligent monitoring nodes will be able to keep track of several temperatures, power supply voltages, fan speeds, and various indicators of system activity. In addition, they will have the ability to control a monitored system under the direction of an administrator sitting at a web browser. As of this writing, much of the initial research and architectural planning is complete. One prototype board has been built and shown to function as expected, and most of the required development tools and licenses have been procured. The hardware design for the first pilot/production board is largely complete. It is expected that these first boards will be built and assembled by late 2000, and software development for this project will extend into 2001.

Context and Motivation

Origins

The Automation and Research Computing Section (ARC), part of the Division of Research and Statistics (R&S) at the Federal Reserve Board (FRB), runs a reasonably extensive computer network for the staffs of R&S and of the Division of Monetary Affairs (MA). There are a total of about three hundred fifty persons in the user community. This "R&S/MA Research Computing Network" consists of more than five hundred computers and over one hundred other network addressable devices – switches, routers, printers, etc.

This network was installed in 1987, and consisted then primarily of Sun 3 servers which were scattered throughout the office areas in the supported divisions. Most users had VT220-type terminals on their desks, while a few researchers had their own Sun workstations. Over time, most desktop devices were upgraded to X terminals, but the servers – which were regularly upgraded and steadily increased in number – remained located in the end-users' office space.

Change

In 1997, the Board's central IT division replaced an aging mainframe and disk farm with newer models which took up a miniscule amount of space compared to the previous system. The then-barren raised-floor space was made available to "client divisions" as a place to put all the servers that were scattered hither and yon. R&S obtained space and resources sufficient to support thirty-five nineteen inch machine racks.

In response to this and other challenges, a complete re-design and re-implementation of the research network was begun, a project which included the migration of all servers to the newly acquired data

center space. This plan was carried out over a two-year period and was completed in mid-1999.

Repercussions

With this change, however, some of the remote diagnostic and control capability inherent in the previous design had been lost. Many times the telephone had been a very effective system monitoring tool – users, strategically placed near servers, were often very effective in providing notification of problems such as noisily failing fans and disk drives, even without much in the way of training. Also, if a system was so sick that it would not respond to the network, these users were often able to reset that system on request. This was especially useful on weekends when support staff were answering calls from home. The data center, however, being a secure "glass house" environment, was off limits to the users and thus this was no longer an option.

The new environment also resulted in some new types of problems. For example, we learned that, while one can equip a storage system with redundant power supplies, this is of limited use if there are no automatic mechanisms in place to provide notification of a failure of one of the supplies. We also learned that the white noise in a data center, together with glass rack doors, can do a remarkably good job of masking even the piercing sound of a piezo-electric alarm two rows away. Finally, the ability to monitor AC power was lost when a new building engineer declared standalone UPSs a fire hazard – the Fire Department needed a way to shut down the entire data center with a single switch, and the distributed UPSs made this problematic. A large, data center-wide UPS was installed in the sub-basement, but the ability to monitor individual circuit loads and power conditions was lost.

It became apparent that something needed to be done in order to gain better status information.

Looking for solutions

To recover our lost capabilities, we determined that a network-accessible instrumentation (or hardware monitoring and control) system was needed. While it is tempting to try to build a single system that addresses every problem, there are many advantages to breaking the problem up into a number of smaller bits that could be addressed individually.

The No-brainer Stuff

On this basis, we saw three areas which would have high immediate impact, were reasonably inexpensive and easy or at least straightforward to do, and which addressed areas which resulted in severe support difficulties. These three areas were serial console access, power control and graphical console access.

Serial Console Access

Several of the devices installed in these racks had serial consoles to which it was fairly straightforward to provide remote access; the large number of serial ports, however, posed difficulties.

Sun SPARC servers

All of the production Sun servers, about 60 in total, are built from Sun Microelectronics motherboards – either the quad-processor UltraAXmp or the uniprocessor UltraAXi. One particular issue with regard to Sun SPARC machines and their serial console is that, by default, these systems will halt when an RS-232 break signal is detected.

As most people who work with Suns know, when one connects a serial console cable to the Sun, the resulting noise on the port often will generate a break condition. While the Sun's halt-on-break behavior can be disabled in the Open Boot PROM, this comes at the cost of not being able to halt a system when required. By contrast, our goal was to be able to connect and disconnect terminals at will and still maintain the ability to halt the system from the console. To solve this problem, a special "non-aborting" console cable from NuData – now part of MicroWarehouse [44] – was installed on each Sun server.¹

CMD RAID Controllers

All of the RAID subsystems are again integrated in-house. We have approximately fifty RAID boxes in production, split between the SPARC and NT servers. The controller used in these subsystems is the CMD [14] CRD-5440. All configuration of this controller is

¹This device, – NuData part number NUD4273 – is a short console cable, one end of which contains circuitry to ensure that a break signal is never generated at the Sun's console port except when one is clearly being generated at the terminal end of the cable. Inspection of the circuit in this device reveals that it is actually fairly complex. It contains fifteen resistors, eight diodes, three transistors, one op amp and five capacitors – one of them a large 1000uF electrolytic.

done via a serial console connection on the back of the unit; the host knows not that it has a RAID controller attached.

Serial port cabling infrastructure

The first implementation of a system to connect to these serial ports, over one hundred of them, was based purely on 25-pair, Cat 3 Telco trunk cables and USOC patch panels. A patch panel was installed in every rack or two, and conventional, straight-through 10-base-T patch cables were used to connect from those to custom-made serial null modem adapters [23]. The other ends of the trunk cables were brought into USOC patch panels. A few standard VT220-type terminals were installed on rack shelves; these could then be plugged into any port in those panels, switchboard operator style.

Serial port server

The switchboard system worked quite well, but the ultimate goal was to be able to connect to any of these ports remotely. To accomplish this, an X86 Linux server was built and outfitted with Cyclades [18] multi-port serial cards. Shell scripts driving C-Kermit [16] were written to provide command-line-based access to the piles of serial ports provided by the Cyclades cards. In this way an administrator on the Linux serial port server could simply connect to the correct console without knowing which serial device it was on. This system could be used either at the Linux system's console or through a secure network connection from anywhere in the world. The multiuser capability of the Linux box made it possible for several administrators to be working on separate systems at the same time.

This of course represents something of a single point of failure in the support infrastructure for these machines; the backup system remains the old switchboard technique, since the Linux/Cyclades system uses the same patch panels.

Power Control

Each of the thirty-five machine racks was outfitted with a dedicated, 20A 120VAC circuit. To help manage these, Model 3302F intelligent power controllers from Pulizzi [56] were installed in each rack. These devices each have eight programmable power outlets, which can individually be turned off or on and can be programmed to turn on in a specific sequence with programmable time delays. These controllers can be daisy-chained on an RS-485 serial network, which then can be connected to a standard ASCII terminal or terminal emulator.

This serial network was connected to the Linux-based serial port server described in the previous section, and scripts were written to interface to that serial network and generate the correct commands to, for example, power cycle a specific port.

Graphical console access

Windows NT, by default, does not support management through a serial console in the same way as

do all these other devices. At present, we have about twenty-two Windows NT servers in production, but no room to put twenty-two monitors. To consolidate access to these devices, a cascaded-KVM switch solution from Cybex [17] was procured.

The primary disadvantage of this solution was that, much as was the case with the switchboard system for the serial consoles, it still required that someone go to the data center to use it. A significant enhancement to this system, based on the "VDE/200" device from Lightwave Communications [35], is currently under evaluation. The expectation is that we will be able to use this device to deliver KVM signals to the desktops of the systems administration staff; as of this writing, we have a single, working control console on the same floor as our offices, across the street from the data center.

Still, the high bandwidth requirements of a graphical console reduce the chances of getting access to the NT server consoles from outside of the two-building local campus. One possible alternative is to provide some sort of serial console access to an NT machine. Pieces of such a solution seem to exist (see e.g. [26]) but it is unclear at this point whether this can be made to work with NT in any satisfactory manner.

Software Monitoring

Of course, most (but not all) of these devices have some ability to be monitored via software, usually within the context of the operating system running on the device. Both Solaris and Windows NT have SNMP interfaces and a great deal can be done with advanced network and system management systems. In addition, Solaris provides a fair amount of event data through syslog, as does Windows NT through the event log.

Many of the monitoring tasks described in the latter portions of this paper could in theory be accomplished through software-based monitoring on many of these systems. For example, Sun's Advanced System Monitoring [63] package provides driver-level access to arbitrary I2C-based [55] monitoring devices, including the various monitoring probes that are provided on the system board itself. It should be a straightforward effort to map these into an extended SNMP MIB and accessed through the UC Davis SNMP daemon [65] that is used on our Solaris machines, or also possibly through CGI scripts run by Apache.² Additionally, many newer X86 server systems support the new Intel Corporation Advanced Configuration and Power Management Interface [29]. ACPI – follow-on to the Advanced Power Management Interface (APM) – seems likely to enable the collection of a wide variety of system information.

²Sun also offers a package called Sun Management Center [61], formerly known as SyMON. This is a relative powerful package; however, it is unavailable for the UltraAXmp and UltraAXi boards

Still, there are three essential difficulties with these software-based approaches:

1. The software work that has to be done tends not to be particularly portable from machine to machine or from OS to OS.
2. All these things tend to change at the whim of the system vendor. It seems to the author that, while by leveraging industry standards such as this, one can have a fairly flexible and functional system in a reasonable amount of time, this often comes at the significant cost of system longevity and compatibility among systems from overlapping generations.
3. These capabilities are all very dependent on the proper functioning of the hardware and operating system being monitored. Generally, this means that these interfaces tend to be more useful for routine monitoring and system configuration changes than for troubleshooting or fault recovery.

The Harder Stuff

Still, it was apparent that, even if all these "easy" tasks were completed, there would remain gaps in the available monitoring and control capabilities, and that closing those gaps was going to be substantially harder.

In-band vs Out-of-Band

It seemed clear that a monitoring and control system would be more useful if it was external to the devices being monitored – in other words, if the system was designed to function out-of-band. Malfunctioning computer and networking equipment cannot be counted on to be functional enough to respond to management queries and requests. This observation was realized by problems we experienced in our early attempts to manage the Windows NT servers: it seemed that one of the first stages in almost any failure mode was to stop talking to the network.

The in-band nature of all the approaches in the previous section (with the exception of the power controllers) is a major limitation. For example, most ATX motherboards by default will only enter standby mode when powered up. This is by design and, even though this behavior can be changed in the system BIOS, there are very good reasons for the default setting – one cannot assume that the power will be stable immediately after it is first restored. Thus, even though the power controllers can reset the power to an NT server, this is only the first step in restoring the server to full function – someone must still push the "on" button.

Supporting Heterogeneity

While many modern computing and communications devices incorporate some monitoring and control capabilities, the implementation of these tend to be highly variable from system to system; monitoring and control procedures and software must be developed separately for each type of system. If one were to deploy, for example, a homogeneous network

consisting of nothing but Sun UltraEnterprise or Compaq ProLiant servers, one could probably design a fairly successful monitoring system that depended on the specific capabilities of those devices.

However, in our research network, such an approach was not an option. As we started this wholesale network redesign, we had made a commitment to acquire most of our computer systems, including both SPARC/Solaris and X86/Windows NT servers, by integrating OEM parts in our own facility. While high-end systems such as those from Sun and Compaq are potentially more reliable, they are substantially more expensive than those which we build in-house – twice as expensive in many cases. Our assessment of this trade-off was that, without substantial budget increases, the “more manageable” systems could not be deployed in sufficient quantities to meet the computing demand. As a result, our production systems do not generally come with some of the advanced software tools and on-board diagnostics that might come with top-of-the-line, commercially integrated systems.

Not Everything is a Nail

The data that may be collected from the various systems, besides being variable from system to system, is often inadequate for many purposes. For example, the Sun UltraAXmp motherboards are able to provide a threshold warning when the temperature reaches a certain level (recorded by Solaris in syslog). If the system gets much warmer, it will shut down. While this is useful, it would be more useful if it were possible to obtain a steady reading of system temperature, that could be routinely recorded into a database for trend analysis.

Also, most systems limit monitoring to what is happening inside the system itself, while it is frequently the case that one would be just as interested in what is going on outside the system. For example, rack ambient temperature and airflow can be very useful in identifying patterns in system problems. Many systems can report information concerning power supply status, but this is usually limited to the DC side of the supply; it is rare that a system will provide information concerning the AC side of the supply: current and voltage, circuit loading, power sags and spikes, etc. Many of our most difficult-to-solve problems occur as a result of failures or deficiencies in systems, such as the AC power supply, which are controlled by other parts of our organization. Obtaining good data on those can often be essential to a resolution.

Finally, much of the instrumentation – both for monitoring and control – provided out of the box with typical computer systems is designed to be useful only to an operator physically present at the system. This is, for example, the case with the power and reset buttons, as well as the indicator LEDs on the system front panel. It is rare that the system manufacturer will provide any straightforward mechanism for remote access to this native instrumentation.

Design

Thus, in parallel with the efforts described in the previous section, an investigation was begun into the feasibility of creating a monitoring and control system which operates separately from the computer systems themselves.

Design Criteria

In approaching this problem, several design criteria were identified:

- Relatively low cost. It did not seem practical, from a budgetary standpoint, for the monitoring system to be comparable in cost to the systems being monitored.
- Small size. Even with thirty-five machine racks, rack space was highly constrained. If these monitoring devices could be made either to fit inside the monitored devices, or fit comfortably in the free space at the back of the racks, they would be much easier to install.
- Low power with no special cooling requirements. In particular, it would be most useful if it were possible to run these boards during time periods when the monitored systems are not able to operate. Optimally, the instrumentation network could be run for at least a couple of days just on battery power. Note that these criteria imply that the monitoring board cannot be powered by the monitored system's own power supply.
- Remotely accessible, preferably using a standard interface that is available on virtually any computer. Ideally, the system would be accessible via a web browser, but access via a character terminal interface would also be acceptable.
- Distributed, modular intelligence – small, single-board intelligent probes, networked together and under the control of a higher-level supervisory system. Each probe board would be dedicated to a single monitored system in order to minimize cross-dependencies among systems.³

Goals

As the whole point of this project is monitoring and control, it is well to take a look at what it is expected that the system will monitor and control, and why.

Monitoring

- Temperature. Each monitoring board should have interfaces to measure four or more temperatures. One could be measured by a chip on the surface of the monitoring board itself, while the other three should be in the form of headers

³The primary alternatives to this would appear to be (a) to have a smaller number of much more powerful systems, each monitoring a relatively large number of basic sensors, or (b) to have a network of individual, intelligent sensors. For a variety of reasons, these were rejected fairly early on in the project.

that will allow the connection of remote sensors. These sensors could be placed on CPU heat sinks, disk drives, outside of the system, etc. The author views temperature as the single most important indicator of system health. If anything is seriously wrong with a system, chances are good that the temperature will either go up or go down.

- DC Power supply voltages. The monitoring board should have headers that can be connected to contacts on the motherboard which carry +12V, +5V and +3.3V. Drift or instability in these values can indicate a weak power supply.
- Fan rotation. Many fans today have a tachometer lead. On each rotation of the fan, this lead is shorted to ground a number of times (four is typical) Feeding a voltage to the tachometer wire through a pull-up resistor will result in a square wave with a frequency that is directly proportional to the fan's RPM. Typically a computer system board will provide this rate counter functionality itself, but it is straightforward to tap into this signal and also do the counting on another board. The monitor board should be able to keep track of at least two fans.
- LED states. When confronted with a malfunctioning system, many people will take a look at the hard drive activity LEDs on the front panel as a quick check to see if there is any disk activity. If the light stays on solid, it can be an indication of a hung SCSI or IDE bus. If it is off solid, this can be an indication that the operating system is not running any tasks. If it is blinking rapidly, it would appear that something is going on – often, by the blink rate, just what is happening on the system can be deduced. The monitoring board should have connections to monitor the state of up to three LEDs. It should be able to report the current on/off state as well as a current count of off-to-on transitions.
- AC current. There are two places at which it would potentially be useful to monitor AC current: At each circuit and at each device. At the circuit level, a current reading will give one a sense of how close a circuit is coming to capacity, and how much more can be added to it without concern. Monitored over time, one potentially can detect, for example, that a new device has been added to a circuit, or that one was added during the night and then removed (those darn cleaning crews!). With a little more work, it would also be possible to monitor voltages, and look for spikes and overall power quality. At the device level, one could additionally detect, on a per-device basis, sudden increases and decreases in current demand that could be indicative of a number of system level

events such as impending failure, shutdown or reboot.

The monitoring board should be able to monitor a single AC current at the device level. We determined that the task of monitoring AC at the circuit level had a number of complexities that made it more appropriate to leave to a later project.

- RS-232 serial interface. The monitoring board should have a minimum of two RS-232 serial interfaces in addition to some sort of network interface. In some cases, such as with the CMD RAID controller, useful information such as the state of a RAID set is available only through a serial interface. It also may prove practical to interface these devices to the serial console ports on the SPARC servers and network switches, but with the serial console network discussed above, this is left for another day. One serial interface would be used for these applications, while the other would be left available as a debugging and local control interface.
- Generic digital inputs. There should be four or more generic digital inputs to account for platform specific capabilities, for example to sense alarm conditions that are provided by some chassis and system boards. At least four such inputs are known to be required to monitor our RAID subsystems. One possible use for such an input would be to create a sort of “watchdog timer”. One might write a daemon or NT service which toggles the state of one pin on the parallel port every five seconds or so. The monitor board could watch for that change and, if it does not occur for twenty seconds or so, set a flag which could be picked up and be reported in an alert. In embedded systems, a timeout on the watchdog timer is usually programmed to result in a system reset; with some caution, this approach could be taken for the monitored system via the relay outputs mentioned below.
- Generic analog inputs. There should be two or more generic analog inputs, again to measure platform-specific values, which could be additional temperatures, airflow, humidity, etc.

Control

- Relays. There should be two relays on the monitoring board, each of which can be used to close a switch header on a motherboard, such as those for the reset or power on/off functions. These should be specifically designed to not close except under tightly controlled circumstances. Of special concern is during monitoring-board power-up and when the relays are connected or disconnected from the monitored system; the monitor board should never cause the monitored system to shut down or reboot.
- RS-232 serial. The same interfaces that were described under “monitoring” can clearly be

used in a control application; for example, a command could be added to shut down a RAID array before powering off a server.

Taking a closer look

With these goals and criteria in mind, several candidate technologies were examined. But first, let us note what is not available.

The market, as far as could be determined, was barren of products that directly met the goals and criteria listed above. Generally, the industry seemed to be taking the view that computers could monitor themselves just fine. Several computer system vendors had system and network management solutions. But, almost without fail, these solutions depended on the existence of a homogeneous collection of reasonably high-end systems, and/or they functioned in-band with the operating system and primary data network. Although many vendors were consulted, and many had suggestions as to how to approach the task, not one claimed to have a solution on the shelf.

Following is an outline of several technologies which were considered for this project.

PC-104 Single-board Computers

Features

PC-104 is a standard form factor for X86 PC/AT computers, using 4" square boards and stacking pin-and-socket connectors to carry the ISA bus. A wide variety of peripheral and I/O boards are available.

Advantages

The advantage of PC-104 is the use of a standard, familiar PC architecture.

Disadvantages

High cost. A fully-functional system can cost in excess of \$1000. Proprietary aspects often complicate software development. Generic sensor interfaces require external signal conditioning circuitry.

Vendors

Ampro [5], Advantech [2], Technoland.

Single-board Microcontrollers

Features

Small boards configured with standard microcontrollers. Usually provide a variety of digital and analog interfaces, not normally designed for expansion.

Advantages

Moderate cost, well-documented and less proprietary.

Disadvantages

Limited mix of sensor interfaces often restrictive. Generic sensor interfaces require external signal conditioning circuitry.

Vendors

EMAC, Inc. [36], Z-World [71], R.L.C. Enterprises [58].

Industrial Control PLC Networks

Features

Programmable Logic Controllers are programmed with basic state machine logic. Networks of individual sensors and controls are connected back to the PLC.

Advantages

Very flexible; wide variety of controllers and sensors available.

Disadvantages

PLC architecture is very limiting. Individual sensors and controls are large and relatively expensive.

Vendors

Advantech [2], Omega Engineering [53].

Custom-designed microcontroller boards

For better or worse, this is the one approach that seemed to be capable of addressing the bulk of the stated goals and criteria. Among the more popular microcontrollers, PIC microcontrollers from Microchip Technology [41] are tremendously easy to work with, and these were selected for the first attempts at custom design.

The PIC microcontrollers are programmable in their own quirky assembler language or in any of several higher-level languages, most notably C. Programs are written into the PICs with low-cost device programmers. The model which saw the most use in this project was the PIC16C76 [42], a 20MHz, 28-pin device with five analog inputs, 8KB of program memory, three timers and a real programmable serial port. These chips cost between \$5 and \$10 depending on quantity, and the minimal parts one needs to add to have a functional controller system (e.g. crystal, voltage regulator, a few capacitors) cost maybe another \$10. On a board based on a PIC controller, one can easily include all the I/O interface circuitry that one would have needed to design for all the other solutions anyway. Several companies, e.g. MicroEngineering Labs [43] sell prototyping boards for the PIC for \$10 or so – with these one can have a functional controller board with a couple of hours of soldering.

This custom design solution, although perhaps not the one which intuition would select, proved to be by far the most straightforwardly malleable, to be the most cost effective – even when the cost of development tools is accounted for – and the most free of proprietary encumbrances. With this approach, promising, concrete results were obtained in a very reasonable timeframe.

A Word About Networking

Physical layer

Three technologies were considered as a choice for the physical layer: RS-485, Ethernet, and CAN. This in fact turned out to be a fairly straightforward choice, however.

RS-485, which is also basis for differential SCSI and many short-haul modems, is very simple in design. It transmits a differential signal over common copper twisted pair wire, one pair for half-duplex and two pairs for full-duplex. A ground return wire is usually present as well, so an RS-485 network cable will most likely have either three or five conductors. Data can be transmitted over such cables at relatively high rates, over reasonable distances, and with a great deal of noise immunity. RS-485 is usable in a bus topology, allowing the connection of dozens of devices to a single length of three- or five-wire cable. At distances likely to be experienced in a data center, RS-485 can easily approach speeds of 115.2 Kbps, while over short distances – as found in SCSI busses – it can support speeds as high as 10Mbps or more.

Upper Layers

While the selection of a physical layer was fairly straightforward, there were no obvious choices for the higher-level protocol. TCP/IP was out of the question – the software required to implement TCP/IP was much too complex to deal with in an 8-bit microcontroller. There appeared to be no “standard” protocol available; there were several proprietary protocols such as Echelon’s LonWorks [24], but it was also very common for embedded systems developers to “roll their own” protocol. After an evaluation of the available options, the latter approach seemed the most promising.

A Networking Architecture

The outlines of an architecture for the networking of the monitoring boards was developed, although never used for reasons which will become clear. For the benefit of those who may wish to take this less proprietary approach, this architecture is described here.

Master-slave model

In order to reduce the complexity of the code which must run on the microcontrollers, and to eliminate the need to handle collisions, we decided that the microcontrollers would speak only when spoken to. Each RS-485 string would have a single master node which would direct the microcontrollers to return data or carry out a control task. In idle time between requests, the controller would constantly be taking measurements in order to reduce the polling latency.

RTOS with simple dispatch table

Each microcontroller would run a relatively simple Real-Time Operating System, which would handle task scheduling, interrupt, timer and queue management, as well as resource (e.g. memory) allocation. As requests come in off the network, a process running under the RTOS would scan request headers and select out only traffic for that controller. As each such packet comes in, a command token in the packet and any needed data would be parsed out and inserted in an appropriate job queue. A process running under the RTOS, until this point blocked in wait for the queue to

fill, would be woken up and given an opportunity to act on the command.

Commands would be classed as synchronous or asynchronous. In the case of a synchronous command, the master node would do nothing – and the network would be quiet – until the slave node returned the results; a timeout would be set so that a dead node would not hang the network. In the case of an asynchronous command, the master node would wait for a receipt acknowledgement but would then go on with other work, giving the slave node time to finish the command. When the command was complete, the slave node would buffer the results and wait for another poll from the master node before attempting to return the results. A request to transmit previously requested results would of course be a synchronous command, requiring immediate response (or negative acknowledgement) If the results are never requested, the slave node would discard the results when the buffer was needed again.

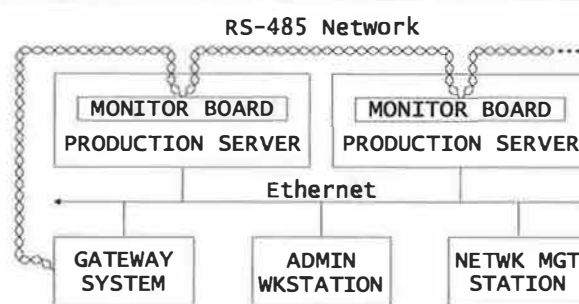


Figure 1: Instrumentation network diagram.

On the Wire

Request packets would contain a preamble, a one-byte target address, a one-byte command token, one-byte request sequence number, an optional data field, a checksum and a trailer. With a single master node, a source address would not be required in a request packet. Response packets would have a slightly different preamble, the source address and response code, followed again by the sequence number, optional data, checksum and trailer. Request packets would need to be acknowledged by the microcontroller within a set time window, but response packets would not require acknowledgement; instead, the master node would simply repeat the request if required.

Start-up node discovery would be done by a simple search through all possible addresses – basic RS-485 networks support only 32 devices and this discovery can be completed in only a few seconds.⁴

Master Node

It is expected that the master node would be a Unix-type machine, probably Linux or BSD. On this

⁴This is an electrical limitation related to the input impedance of the transceivers; high-impedance transceivers are available which will allow up to 128 devices to be attached, but the use of these seems unnecessary in this application

master node, the protocol would be implemented as user-space executables which could be called by CGI scripts from the Apache web server. The service requests and results could be delivered from and to the user via the http protocol. Automatic cron jobs could run on the master node to make routine, periodic requests for data that would then be stored in a database.

Disadvantages

There are some disadvantages of this system; it does not, for example, provide a mechanism for the microcontroller to report alerts or interrupts back to the master node. A non-responsive or ill-behaved slave node would need to be flagged as a potential problem, and an alert generated through the network management system. Also, the number of devices hanging off each serial port would also be limited by how many could successfully be polled without using up all the available bandwidth with overhead processing.

emWare

This architecture was not implemented, however, because a source, emWare [25], was identified for a product that did essentially the same thing. emWare's product, EMIT (Embedded Micro Internetworking Technology), provides a networking protocol, an http gateway, skeleton source code for several microcontrollers, and Java Beans that can be used to create a graphical interface to the networked controllers. It was estimated that that this product would take at least six to twelve months off the time to develop the instrumentation software, and a much more functional end product would result than would have otherwise.

First Working Prototype

Ultimately a wire-wrapped, PIC-based prototype was designed and built. This prototype did most of what was required, including running the EMIT software and talking to a web browser via the Java Beans. The browser window would display the constantly-

updated board temperature, display counts of the HDD activity lights, and a Java-based button could be depressed to initiate a hardware reset on a Windows NT server. Finally, an affordable and buildable solution was in sight. This was in the fall of 1999, and this board was the one that was described to some LISA participants in a Work In Progress session last year.

After completing this prototype, however, it became clear that the PIC had a severe shortcoming: The PIC simply did not have enough data memory, and it had no external memory interface. The most serious problem resulting from this was in interfacing with the CMD RAID controller's serial port. Although the RAID controller had a mode for talking to such automated devices, use of that mode entailed the ability to buffer records that were comparable in size to the PIC's entire memory. Thus, a different microcontroller was going to be needed.

Atmel AVR

After a review of the alternatives, Atmel's [8] AVR architecture was selected, specifically the ATmega103. The AVR was still an 8-bit microcontroller and was still inexpensive, but the instruction set was much more powerful than the PIC's and it had an external memory interface. Over the next few months, a fairly complete board was designed around on this part, while many design details were tested on low-cost development boards that are available from Atmel. At around the end of Spring of this year, the design was nearly complete, and attention was given to the task of procuring the parts that would be required. Especially in a small project such as this, it is generally a good idea to have the parts in hand before the boards are made, so that adjustments can be made in the event that some parts are not available.

The Parts Shortage of 2000

One does not normally, however, expect the availability problem to be as severe as was experienced in this case. Parts were so hard to come by this

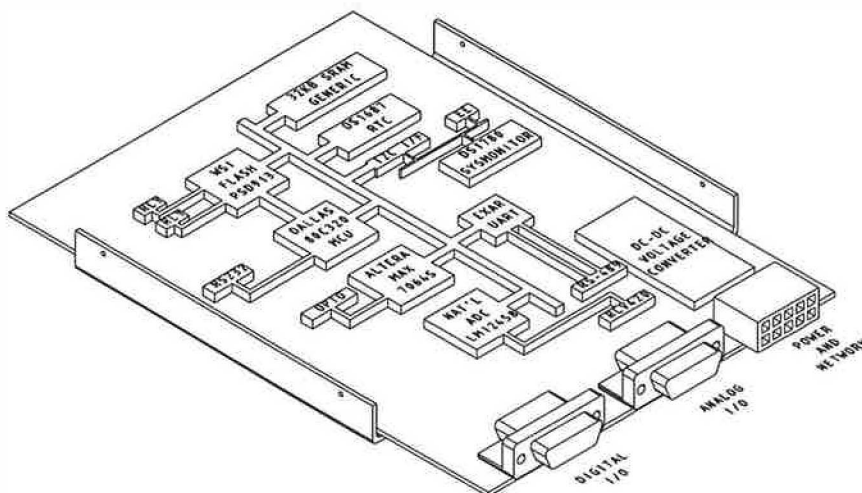


Figure 2: Conceptual drawing of monitor board.

past Summer, that it was headline news every week in publications such as EE Times and Electronic Buyer's News (EBN). Pretty much anything in a surface-mount package, anything with memory in it, or anything that was used with anything that was surface mount and/or used memory – especially flash memory, was simply not available. EBN ran a story about an OEM CEO who raided his employees' Palm organizers just to get the flash memory [52]; see also [69] and [68]. In the case at hand, about half the parts on the board – which was designed using surface-mount technology – were unavailable, most notably the Atmel microcontroller itself. Atmel reportedly was so far behind that they had ceased taking orders, and Atmel distributors were quoting lead times as long as ten months. There was no way around it, it was either shelve the whole thing for a while or go back to the drawing board.

Yet Another microcontroller

Thus, the search began for yet another microcontroller. This time, the search was limited to devices which could (a) use as much of the existing hardware design as possible, (b) were available for immediate purchase and (c) for which there existed a port of the emWare code. Given the scarcity of parts, this turned out to be a pretty short list; very few microcontrollers of any sort were obtainable at that point. In the end, a supply of Dallas Semiconductor DS80C320 chips was located. The DS80C320 is sort of a souped-up, high-speed 8051-architecture microcontroller with no internal program storage. They also have relatively little in the way of built-in peripherals, so the search had to be expanded to include several new parts, such as an external analog-to-digital converter and EEPROM memory. Parts in hand, work was begun to design the board one more time. However, this time around, it was going to have to be larger – the perimeter of a CD-ROM drive rather than of a floppy drive as originally planned – and would have a higher power consumption than desired. But something had to give, and at least work was still progressing.

Working Design

Overview

Almost all of the parts used in this board will be “through-hole” as opposed to “surface-mount”. The board itself will be designed in four layers (two signal layers, a ground layer and a power layer), with the outside dimensions about the same as that of a CD-ROM drive. Mounting holes will be drilled near the edge in such a way as to make it straightforward to mount the board, using angle brackets, in a standard 5 1/4” drive bay. The plan is to mount this device in a free hard drive bay inside the server device. The instrumentation network and power supply for the board will enter the system through one of the I/O slot hanger brackets, and bundles of sensor wires will be connected from various locations within the server to connectors on the back of the monitor board.

A conceptual drawing of the board design is provided in Figure 2.

Power

The boards are designed to operate from a +24V power supply. The relatively high DC voltage is used for two reasons: First, some devices, such as the 4-20mA current loop, need the 24V to operate. Second, the plan is to daisy-chain the power supply alongside the RS-485 network. If the boards were designed to run from a +5V power supply (voltages will need to be converted on-board in any event), the supply current would have to be higher; this would have a number of implications, including the use of heavier wires in the daisy-chain harness. The 24V power is converted to +5V, -12V and +12V on-board using off-the-shelf DC-to-DC converters.

Connectors

All the run-time I/O connectors will be on one of the short edges of the board. The design calls for three connectors. One rectangular nylon connector will carry connections for power input, the RS-485 network and the 4-20mA current loop.

The other two connectors are both 26-pin high-density D-sub connectors. One will carry most of the analog connections, while the other will have the serial ports, relays, counters and other digital connections.

There will also be small header connectors on the surface of the board for the attachment of JTAG cables, for the programming of the two programmable logic parts used in the design.

Components

In this design, there are about twenty or so integrated circuits, and about two or three dozen other parts. Some of the more interesting parts are described here; refer to Figure 3 to see how these parts work together.

Dallas Semiconductor DS80C320

The Dallas Semiconductor DS80C320 [21] is a clone of the Intel 80C32 [30]. Dallas' part uses a new core design, runs faster than the old Intel part (up to 33MHz), and does more in each clock cycle than the traditional 8051 designs. The Dallas part also has an extra serial port, extra timers and a few extra external interrupt pins. This is actually about as nice as 8051-architecture chips get.

Waferscale Integration PSD913F2

Waferscale [67] makes a line of highly-integrated parts that they call “Programmable System Devices”, or “PSDs”; the current design uses the PSD913F2 [66], which provides 160KB of flash memory in twelve sectors, 2KB of SRAM and a 57-input/19-output simple PLD. The PSD provides all the program storage for the board, some of the data storage, and handles all the address decode work needed to generate chip selects for the other ICs on the board.

National Semiconductor LM12458

National calls the LM12458 [48] a “data acquisition system”. It is a highly programmable analog-to-digital converter, with eight inputs. The conversion circuitry can give a result with twelve bits of precision, and it can buffer several results to take some processing load off the microcontroller. It can also be programmed with threshold values, which allows for the generation of an interrupt if a sensor drifts out of specification.

Altera MAX7064S

The Altera MAX7064S [4] is a 64-macrocell CPLD, (Complex Programmable Logic Device). The purpose of this part here is to handle the counting of the three HDD activity LED interfaces. The three LED headers on the monitored system’s motherboard will be connected (via optoisolators and buffers) to input pins on the CPLD. The CPLD increments an internal counter every time one of the counter input pins transitions from low to high. It communicates to the microcontroller through a parallel interface; the microcontroller thinks that the CPLD is just three bytes of memory.

Dallas Semiconductor DS1780

The Dallas DS1780 [20] is a system monitor chip, designed for use on PC motherboards. It is capable of monitoring the surface temperature at the chip itself, as well as two fan speeds, several power supply voltages, and various other parameters. It has a register that latches the state of five external pins on power-up; this is designed to read an identifier off a Pentium processor, but here it is used with a DIP switch to latch the board’s address on the RS-485 network. Finally, the DS1780 has a system reset circuit, which will keep all the chips on the board in a reset state until the power supply has stabilized.

STmicroelectronics M24C32

The M24C32 [59] is a byte-addressable, 32 Kbit serial EEPROM. Although there is plenty of flash and SRAM memory elsewhere on the board, neither of

these are particularly good at storing small amounts of dynamic data which must survive a system power cycle. This device can be used to store various setup, configuration and calibration parameters such as the monitor board’s monitored system name. It can also occasionally be used to log various critical events that one might need to retrieve after the system was power cycled.

Philips Semiconductor PCF8584

Since the DS80C320 does not have an I2C interface, and both the DS1780 and the serial EEPROM required one, the design calls for a Philips PCF8584 [54], an I2C bus controller, to do this job.

Dallas Semiconductor DS1687

The Dallas DS1687 [19] is a battery-backed real time clock (RTC). It will be used by the board both to keep track of the time and to generate a square wave input to the microcontroller which will provide a regular system “tick” for the RTOS, that is used to schedule various housekeeping activities. This part also provides 256 bytes of NVRAM.

Burr-Brown RCV420

The Burr-Brown RCV420 [10] is a 4-20mA current loop receiver. 4-20mA current loops work by supplying around 24V to a pair of wires which are carried out to a remote sensor. The sensor is powered off the loop wires themselves, and communicate back to the receiver by regulating the loop current to be a value in the range of four to twenty milliamps which, net of the 4mA base, is proportional to the value the sensor wishes to report. Current loops can be run for long distances, require only two wires to carry both power and signal, and have excellent noise immunity.

The RCV420 senses the loop current and outputs a voltage proportional to the reported current and in the range of 0-5V, which is what most analog to digital converters are designed to measure. In this board, the RCV420’s output will be connected to the LM12458. The RCV420 will be used to interface to a remote AC current sensor.

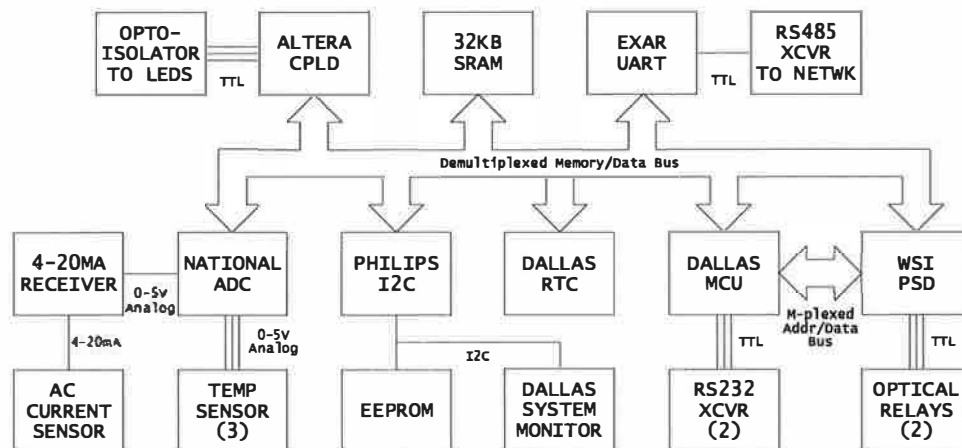


Figure 3: Block diagram of monitor board functional components.

Sensors

Temperature Sensors

In addition to the single temperature measured by the DS1780, three remote temperature sensors will be supported. The remote sensors will use the National Semiconductor LM20 [49], a tiny, 2mm x 1.25mm x 1mm surface-mount part.

AC Current Sensors

Although this was not anticipated, the AC current sensor interface turned out to be one of the hardest parts of the entire design to get right.

It is quite easy to come by remote current sensors which transmit information via 4-20mA current loop – this is a standard type of part used in industrial process instrumentation applications. However, while the specifications of hundreds of toroidal-transformer and Hall current sensors from several different vendors were reviewed, none were identified which simultaneously met the specifications and goals for size (to be able to put one in a regular outlet box, or inside the power controller), frequency response (60Hz), sensing range (1-20A) and cost (<\$50). In addition, for many industrial current sensors, the translation between the AC current and a single value to be transmitted on the 4-20mA current loop is not always well-defined or particularly sophisticated. A sample of one such sensor was disassembled and inspected, and determined to use a particularly simple rectifier and linear integrator circuit which would, at best, give only an “indication” of the current level, not something that could be reliably mapped to an RMS value.

This is particularly an issue with the kinds of loads presented by computer equipment. Unlike resistive loads, which will usually have AC current profiles which look pretty much just like the voltage profile (i.e., sine waves), the switching power supplies used in computer equipment consume AC current in short bursts. This sort of signal is said to have a high “crest factor”; this presents special challenges in the measurement of an absolute AC current level – see [33]. Many industrial current sensors are designed under the assumption of either resistive or inductive (motor) loads; these assumptions do not necessarily translate well into this application. Thus, it is not really fair to say that the device mentioned in the previous paragraph was poorly-designed; rather, it is more that it just was designed for a different application. Errors in current values reported by simple averaging circuits for signals with high crest factors can be as high as 50% or more.

Candidate design

In the current sensor design developed for this project, the transducer is a small, toroidal current transformer made by Coilcraft [15]. This is an inexpensive (\$5 or so) part which is just a little more than ten cubic centimeters in volume. The current output of the CS60 is dropped across a small resistance, and the

resulting voltage is input into an “RMS to DC Converter” – the AD737 [6] from Analog Devices. To translate the output of the AD737 into a 4-20mA current signal, a Burr-Brown XTR101 [11] current loop loop transmitter is used.

While, as of this writing, this circuit design is not complete, it should result in a relatively small package, as current sensors go, and should cost \$50 or less to build. It is believed that the primary way to improve on this design is by use of a high-speed analog-to-digital converter and a digital signal processor chip, which would be considerably more complex and costly.

While the requirement for a 4-20mA current loop receiver adds complexity to the overall design, the availability of such an interface on the monitoring board, because of the wealth of sensors with 4-20mA outputs available for industrial process instrumentation systems, gives this board a great deal of flexibility in being able to handle new and unanticipated applications.

RTOS

While, at this point, emWare’s EMIT software appears to be quite suitable for this project, the skeleton microcontroller code provided has a number of shortcomings. It generally will work, but it is not particularly modular. One would like to see a structure for building modular, cooperative processes calling library routines which implement a well-defined API, but the emWare code is more like a giant, monolithic event loop into which one inserts one’s own code. It is designed sort of like an RTOS, but a number of essential RTOS services are missing, or blurred into the main event loop.

Thus, the plan is to break that code up – much of it may have to be substantially rewritten – and re-implement it as a number of modular tasks under the control of an actual RTOS. This is somewhat less daunting than it may seem, because the existing code does at least provide a good, working example, and a large proportion of the emWare code is not useful in this application (e.g. modem support) or is not relevant in the context of an RTOS.

The working plan has been to use Jean Labrosse’s uC/OS-II as the RTOS. One purchases the source code to uC/OS-II by purchasing Labrosse’s book on the product. What comes with the book is only licensed for non-commercial use and is unsupported in the sense that one has to purchase upgrades and bug fixes.

In researching references for this paper, however, the author learned of a new, fully free RTOS that had just been released in mid-1999. The PROC [50] Real-Time Kernel, written by Jan Erik Nilsen of Nilsen Elektronikk, is a very straightforward RTOS kernel which has been ported to several microcontrollers and would seem to be fully capable of meeting the requirements for this project.

Status

Hardware

As of this writing, the schematics for the board design are largely complete except for some of the analog interfaces. The next major step is to develop the board layout, which amounts to drawing where the chips go on the board and how all the traces get from one chip to the other. When that is complete, the design will be taken to the board fabrication house to have one panel of printed circuits made. A few of these will be populated with parts and tested by hand. If the boards work as planned, further assembly will be moved to a contract manufacturer.

There are some additional boards which have to be made, which are various stages of completion:

- Since most PCs do not have an RS-485 port, an RS-232 to RS-485 converter board [22] was designed. This board works by simply attaching an RS-232 transceiver and an RS-485 transceiver back to back. One panel's worth of these boards have been made, and the samples that have assembled have worked exactly as required. Thus, this board is largely complete.
- There was only one part for this design which was not available in a through-hole form – the Dallas DS1780 – and no similar system monitor chip seemed to be available in through-hole from any other vendor. Thus, rather than have a single surface-mount part on the board, a small adapter board was designed and enough were manufactured to last through the first production run of the monitoring board.
- Boards on which to assemble the remote temperature and AC sensors will also be required; these designs are partially complete.

Software

One of the hardest parts of doing this work is that so little of the software can be written and tested before the hardware design is done. Thus, while an architecture for the software has been developed at a fairly high level, and both the emWare/EMIT and uC/OS-II software have been tested on actual hardware, little has been done of the down-and-dirty work of banging out the code to make this thing all work.

Availability

Neither the author nor his employer have any proprietary interests in the portions that have been developed by us or at our behest. This includes the general system architecture, the hardware designs, including all printed circuit board schematics and layouts. As portions of this work are completed, it is intended that these will be made available on the Internet – of course with no warranty or support. Two portions of the design, the RS232-to-RS485 adapter design [22] and the Null-modem adapter design [23] are available in this way, and others should be available soon, as time allows. This paper is in part

intended to serve a similar purpose. If the reader has a specific interest in some portion of what has been done here that has not yet been made available, please let the author know.

However, since many portions of the project make use of proprietary software tools, there may be some gaps in what can be released. For example, to be able to modify, compile and run the software that will be written for the monitoring board, a potential user is likely to have to obtain licenses from Tasking [64] for the C compiler, and emWare [25] for the EMIT SDK. Both the Waferscale and Altera programmable logic parts were selected in part because their manufacturers provide excellent free development tools.

This being said, one secondary goal in this project has been to avoid the dependence of the hardware components on commercial software packages. It should be quite feasible to build a few of these monitoring boards and write completely different programming for them. Implementations of the services provided by the commercial packages could likely be written from scratch if one is sufficiently determined.

It should also be noted that all these printed circuit board designs are being prepared with the Eagle layout editor, from CadSoft [12]. While these files will not be importable into competing design programs, they will be readable and printable with CadSoft's freeware "lite" version of Eagle, which also serves as a viewer. However, one would not be able to modify the designs with the free version because they are larger and have more layers than the free version will allow to be edited. The professional version of Eagle is, however, quite reasonably priced compared to many other EDA packages. Also, the "Gerbers" will be made available – Gerbers are the native format for the Gerber photoplotters, and have become the lingua franca of the PCB fabrication industry.

Project future

Looking toward the future, the author believes that an optimal outcome would be that this idea – of a instrumentation network, designed in response to requirements specific to a data center environment and which operates quite independently from the production computer and communications systems – will catch on, and we will ultimately see commercial products along these lines, complete with "connection" or "probe" kits for various commercial and generic computer systems, much in the way that KVM switches are sold today. The author hopes that, were this to come to pass, the controller modules themselves will be designed in an open and standard manner, so that users of such equipment will be able to expand the capabilities both of the hardware and of the software.

Toward this end, it seems that one significant contribution that could be made by this project would be to begin to define standards for how such an instrumentation network should operate. In that regard, a starting outline is offered here:

Sensor Interfaces

Some of the hardest parts of this hardware design are the interfaces between the microcontroller and the sensors or controls. While 4-20mA interfaces will suffice for many sorts of sensors, for others they may not be as practical or cost-effective.

Since temperature sensors would be a frequent choice in this application, it would make sense to optimize an interface just for these. Temperature sensors vary considerably in their electrical interfaces. For example, thermistors are devices which vary in resistance over temperature, thermocouples generate tiny, temperature-dependent voltages with response curves that are quite dependent on the metals used, while some IC sensors, having pre-linearized voltage outputs, vary by 10mV per degree. The interface requirements for these three types of sensors are quite different. The differences are, however, sufficiently small that it would be fairly simple to specify one or two standard temperature sensor interfaces for a computer monitoring board which could support the connection of any of these types of sensors through a simple conditioning circuit. A four-pin connector, for example, carrying +5V supply, supply ground, and a differential analog signal return, or a five-pin connector with +12V, -12V, supply neutral and the differential return, would likely work quite well for such an application. By defining these interfaces, the board design becomes more straightforward, and standard sensor designs, covering a wide range of functionality, could be defined and mass-produced.

Controller Hardware Architecture

While this may not be the most practical area for standardization, the creation of one or two reference designs can be very useful in creating a standard which can be widely implemented. An important consideration here will be the selection of an architectural level for the microcontroller – 8-bit, 16-bit or 32-bit. This choice will have a great impact on many other design decisions, such as the RTOS or the networking protocol. It will also determine much with regard to the overall cost and complexity of the design. The author's personal assessment is that this application calls for an 8-bit architecture, although a 16-bit processor, while possibly overkill, might be considered if strongly indicated by other considerations. 32-bit architectures would appear to be excessive at best.

Controller RTOS

The RTOS, or Real-Time Operating System, that would run on the monitor boards is something that would be very useful to standardize. An important criteria for RTOS selection is likely to be the openness, or "freeness" of the RTOS architecture and implementation both for commercial and non-commercial use. There exist several RTOSs which are "free" in the sense that one can download the source and tinker with it, but which are otherwise restricted in ways which limit their usefulness for an open, collaborative project. uC/OS-II [40], for example, has this problem.

There are also several RTOSs which are fully free in the sense required, but which are designed only for 32-bit or wider processors. Some examples of these are RTEMS [51], eCos [57] and the various embedded (although in some cases not fully Real-Time) Linux versions (Embedix [37], uClinux [38] and Hard Hat Linux [45]).

Given an assessment that 32-bit architectures are uncalled for here, this would imply a need for a free RTOS targeted to smaller architectures. Unfortunately, these seem to be somewhat less common. Three RTOSs meeting this criteria have been identified by the author: Uros Platise's UROS⁵, Nilsen Elektronikk's PROC [50], and Kate Alhola's Katix [3]. Another, Barry Kauler's Screem [31] may also be sufficiently free for such a project. PROC appears to be the most interesting, complete and portable of these.

Driver Interface

Within the context of a selected RTOS, the concept of a sensor driver module could be defined, leading to significant opportunities for code reuse.

Networking Protocols

A networking protocol would of course be crucial. While searching for information in support of the research for writing this paper, it became clear that the state of – or at least the state of information on – higher-level protocols for CAN has improved considerably since the early stages of this project. It seems likely that CAN, together with some reasonably open higher-level protocol, would make an excellent choice for a networking protocol in a standard for this kind of instrumentation system.

The difficulty with CAN is that, like Ethernet and to a lesser extent RS-485, CAN is a specification at the physical and data link layers of the OSI stack. These layers typically are implemented in silicon, either as an on-board peripheral in a microcontroller, or as an external interface chip. This means that, in order to be useful, a CAN design must include an "HLP", or Higher Level Protocol. Unfortunately, there are literally dozens of HLPs to pick from, and most are proprietary, incompatible with one another and highly application-specific. This has also resulted in a great deal of fragmentation in the CAN market, to the point that it is not clear to this author that any complete, fully open, free software implementation of a CAN HLP stack exists. For more information on CAN and the tangle of HLPs, see [34], [39], and [32].

Nonetheless, this must be moderated by the practicality of implementing such a protocol within the context of the selected microcontroller and RTOS. The PROC Real Time Kernel includes an architecture for a basic multi-processor message-passing interface which may prove sufficient for an application such as this.

⁵During the preparation of this paper, Mr. Platise's website seems to have gone off the air. The author does not know of a mirror, and hopes that it will return soon

Gateway architecture

To provide an interface between the system administrators and the instrumentation network, some sort of gateway system needs to be defined. As mentioned earlier in this paper, the original design called for the use of an HTTP server, probably Apache, to provide this interface. The construction of simple CGI programs to interface to the back end network, or even, perhaps, an Apache module (`mod_instrumentation?`) would be a very straightforward way to accomplish this. Another possibility, of course, is to use an SNMP agent to fill this role. A collaborative project to define a standard for data center instrumentation systems could, as part of this effort, define an SNMP MIB for the various sorts of interfaces which one would expect to find on the monitoring probes. This would of course provide for excellent integration between the instrumentation network and existing network and system management tools.

Conclusions

While it is difficult, at this point, to conclude that the results have been worth the effort, the promise of what it should be able to do once complete, and the progress that has been made, have been very encouraging.

Despite the need to make a substantial investment in specialized equipment and software for this project, these costs remain modest compared to overall hardware costs for a network such as this. For example, a basic, Sun Enterprise 420R with a single processor and 1GB of memory has a list price of about \$25,000, whereas a comparable machine built from an UltraAXmp motherboard will cost about \$13,000; the marginal costs of upgrading these systems will continue to favor the UltraAXmp, since, as we are acting as an OEM, the additional parts are normally purchased from distributors rather than VARs. Even with the discounts that can be expected for the 420R, the savings here can easily reach \$10,000 to \$20,000 or more per system. The situation for X86 servers and RAID subsystems is similar. Extended over dozens of RAID-equipped servers, the total advantage, in terms of acquisition costs, of the in-house integration approach is clearly on the order of hundreds of thousands of dollars. By contrast, the total investment in development hardware and software has been on the order of tens of thousands of dollars. Additionally, when the design is complete, the per-system implementation costs should be well less than \$500.

Acknowledgements

Clearly this project – specifically the design of the monitoring board and associated network – has taken much longer and has been much more complex than had originally been anticipated. The author is extremely grateful to his division and supervisors at the Federal Reserve Board for having allowed him the latitude to pursue this for so long.

The author would like to acknowledge the assistance of three people in this project and in the preparation of this paper. First, the author's intern in the Summer of 1999, Pete Owen, put together the PIC-based prototype, complete with getting the emWare code to run and the Java interface to work. Spencer Bankhead, a member of our systems staff, has done the bulk of the work on the console and power controller networks described early in this paper. Spencer is also responsible for getting the author roped into writing this paper. Finally, the author would like to thank Doug Hughes for invaluable assistance in the preparation of this paper.

Author's apology

During the final stages of editing this paper, work unfortunately came to an abrupt halt as the author developed a badly infected gallbladder that had to be removed on an emergency basis. Although the author did have an opportunity, following the surgery, to tie up some loose ends, he believes that there remain some significant shortcomings that would have been corrected had time and health allowed. The author is indebted to Rob Kolstad and Doug Hughes for their assistance in making changes at the last-minute.

Author Information

Bob Drzyzgula is a native of central New York State. In 1978, he graduated in with a BA in Math and a BA in Physics from the State University of New York at Oswego. From 1978-1980, he studied mathematics at the University of Virginia in Charlottesville. After working in the private sector for three years, he joined the staff the Federal Reserve Board in 1983, where he is now responsible for network and computer systems architecture and planning for the divisions of Research and Statistics and Monetary Affairs.

The author may be contacted at the Internet address bob@frb.gov, or at the postal address: Automation and Research Computing Section, Division of Research and Statistics, Federal Reserve Board, 20th & C streets, NW, Washington, DC 20551. Views expressed herein do not necessarily represent those of the Board of Governors of the Federal Reserve System.

References

- [1] Advanced Micro Devices, *Embedded Processors*, <http://www.amd.com/products/epd/index.html>.
- [2] Advantech Company, Ltd., <http://www.advantech.com/>.
- [3] Alhola, Kate, <http://www.funet.fi/kate/katix.html>.
- [4] Altera Corporation, *MAX7000S Series CPLD*, San Jose, CA, <http://www.altera.com/html/products/m7ks.html>, August 11, 2000.
- [5] Ampro Computers, <http://www.ampro.com>.
- [6] Analog Devices, *AD737*, <http://products.analog.com/products/info.asp?product=AD737>.

- [7] Arcom Controls, *Target188 spec*, <http://www.arcomcontrols.com/products/icp/pc104/processors/Target188.htm>.
- [8] Atmel Corporation, <http://www.atmel.com/>.
- [9] Bridges, Patrick, *Current Operating Systems Projects and OS-related research*, <http://www.cs.arizona.edu/people/bridges/oses.html>.
- [10] Burr-Brown RCV420 data sheet, Burr-Brown Corporation, Tucson, AZ, October, 1997.
- [11] Burr-Brown XTR101 Data Sheet, Burr-Brown Corporation, Tucson, A, October, 1993.
- [12] CadSoft Computer GmbH, <http://www.cadsoft.de/>.
- [13] CAN in Automation International Users and Manufacturers Group (CiA), <http://www.can-cia.de/>.
- [14] CMD Technology, <http://www.cmd.com/>.
- [15] Coilcraft, *CS60-50 Current Sensor home*, <http://www.coilcraft.com/sen60t.html>.
- [16] Columbia University, *Kermit Project home*, <http://www.columbia.edu/kermit/>.
- [17] Cybex Computer Products Corporation, <http://www.cybex.com/>.
- [18] Cyclades, <http://www.cyclades.com/>.
- [19] Dallas Semiconductor, *DS1687 Data Sheet*.
- [20] Dallas Semiconductor, *DS1780 Data Sheet*.
- [21] Dallas Semiconductor, *High-Speed Micros*, <http://www.dalsemi.com/products/micros/hispeed/index.html>.
- [22] Drzyzgula, Bob, *Design for an RS232 to RS485 converter*, <http://www.drzyzgula.org/bob/electronics/circuits.shtml>.
- [23] Drzyzgula, Bob, *Design for Modular Null-Modem Adapters*, <http://www.drzyzgula.org/bob/electronics/nma.shtml>.
- [24] Echelon Corporation, <http://www.echelon.com/>.
- [25] emWare, <http://www.emware.com/>.
- [26] Enitek, *J1 Serial Console Card*, <http://www.enitek.com/j1/>.
- [27] Forth Interest Group, <http://www.forth.org/>.
- [28] Intel Corporation, *80186 processor*, <http://developer.intel.com/design/intarch/intel186/>.
- [29] Intel Corporation, *Advanced Configuration and Power Management Interface*, <http://developer.intel.com/technology/iapc/>.
- [30] Intel Corporation, *MCS51 Architecture*, <http://developer.intel.com/design/mcs51/>.
- [31] Kauler, Barry, *Goosee*, <http://goofee.com/goosee/index.html>.
- [32] Korane, Kenneth J., "Mobile Machines get CAN in Gear," *Machine Design Magazine*, <http://www.canhug.org/info/articles/machdes.htm>, September 12, 1996.
- [33] Kosbar, Kurt, *Basic Time Domain Measurements*, <http://www.siglab.ece.umn.edu/ee301/dsp/basics/dpe.html>, 1998.
- [34] Lawrenz, Wolfhard, *Worldwide Status of CAN – Present and Future*, http://www.fh-wolfenbuettel.de/fb/i/organisation/institut_fuer_verteilte/CAN/canfund.htm.
- [35] Lightwave Communications, Inc., <http://www.lightwavecom.com/>.
- [36] EMAC, Inc., <http://www.emacinc.com>.
- [37] Lineo, *Embedix*, Web page. http://www.lineo.com/products/realtime_linux/index.html.
- [38] Lineo, *uClinux*, http://www.lineo.com/products/embedix_uclinux/index.html.
- [39] Marsh, David, "CANbus ICs marry mechanics with electronic supervision and control," *EDN Magazine*, http://www.ednmag.com/ednmag/reg/1997/030397/05DF_EUR.htm, March 3, 1997.
- [40] Micrium, Inc., <http://www.ucos-ii.com/>.
- [41] Microchip Technology, <http://www.microchip.com/>.
- [42] Microchip, *PIC16C76 Data Sheet*, Chandler, AZ, 1997.
- [43] MicroEngineering Labs, <http://www.melabs.com/mel/picproto.htm>.
- [44] MicroWarehouse, <http://www.warehouse.com/>.
- [45] MontaVista Software, <http://www.mvista.com/>.
- [46] Motorola Semiconductor Products, *Microcontrollers*, <http://www.mcu.motsp.com/>.
- [47] National Institute of Standards and Technologies, *Hall Effect Measurements*, <http://www.eeel.nist.gov/812/hall.html>, July 20, 2000.
- [48] National Semiconductor, *LM12458*, <http://www.national.com/pf/LM/LM12458.html>.
- [49] National Semiconductor, *LM20 temperature sensor*, <http://www.national.com/pf/LM/LM20.html>.
- [50] Nilsen Elektronikk AS, *PROC Real Time Kernel*, <http://www.nilsenelektronikk.no/neproc.htm>.
- [51] OAR Corporation, *RTEMS*, <http://www.oarcorp.com/RTEMS/rtems.html>.
- [52] Ojo, Bolaji and Claire Serant, "Desperate Measures," *Electronic Buyer's News*, <http://www.ebnews.com/story/OEG20000519S0032>, May 19, 2000.
- [53] Omega Engineering, <http://www.omega.com/>.
- [54] Philips Semiconductor, *PCF8584*, <http://www.semiconductors.com/pip/PCF8584P/>.
- [55] Philips Semiconductor, *I2C Technology*, <http://www.semiconductors.com/i2c/>.
- [56] Pulizzi Engineering, Inc., <http://www.pulizzi.com/>.
- [57] Redhat, *eCos*, <http://www.redhat.com/products/ecos/>.
- [58] R. L. C Enterprises, <http://www.rlc.com>.
- [59] ST Microelectronics, *M24C32 Serial EEPROM data sheet*, December 22, 1999.
- [60] Stanley, Paul, *Lecture Notes on the Hall Effect*, <http://www.physics.orst.edu/ph213/lecture/l2/node1.html>.
- [61] Sun Management Center, <http://www.sun.com/software/sunmanagementcenter/docs/index.html>.

- [62] Sun Microsystems, Inc., <http://www.sun.com/>.
- [63] Sun Microsystems, *Advanced System Management Manual*, Palo Alto, CA, 1999.
- [64] Tasking, Inc., <http://www.tasking.com/>.
- [65] UC Davis, *SNMP Project*, <http://ucd-snmp.ucdavis.edu/>.
- [66] Waferscale Integration, *PSD8xx/PSD9xx*, <http://www.wsipsd.com/html/PSD8xx9xx.html>.
- [67] Waferscale Integration, <http://www.wsipsd.com/>.
- [68] Wallace, Hank, "Coefficient of Frustration at an All Time High – Reader Response," *Chipcenter*, <http://www.chipcenter.com/eexpert/hwallace/hwallace026a.html>, 2000.
- [69] Wallace, Hank, "Coefficient of Frustration at an All Time High," *Chipcenter*, <http://www.chipcenter.com/eexpert/hwallace/hwallace025a.html>, 2000.
- [70] Western Design Center, <http://www.wdesignc.com/>.
- [71] Z-World, Inc., <http://www.zworld.com/>.
- [72] Zilog, Inc., <http://www.zilog.com/>.

Improving Availability in VERITAS Environments

*Karl Larson – Tellme Networks
Todd Stansell – GNAC*

ABSTRACT

Demands for high availability are increasing almost as fast as storage and performance requirements, posing seemingly impossible challenges for system administrators. VERITAS provides a variety of tools aimed at overcoming these obstacles, but they're not often used effectively.

We'll show you some tools and techniques that work with VERITAS products to help you evaluate whether your systems are already overloaded, and what you can do to stall for time. We'll describe some environments particularly prone to storage-scaling issues and suggest changes you should make before crisis strikes. Finally, we'll provide some suggestions for decreasing the consequences of failures not preventable through fault tolerance.

How To Tell If You're Already Screwed, And Some Quick Fixes

Gather Layout Information

In order to make intelligent changes to improve your storage environment, you must collect detailed performance and configuration information. Most of the tools needed to collect this information are probably already installed on your systems – you just need to use them. We'll also mention a few other tools and scripts that we and others have created to collect and present this data. Details on how to obtain these third-party tools are included in the appendix.

The most important initial steps with VERITAS Volume Manager (VxVM) installations are obtaining information about and understanding your disk layout. The provided utilities do a poor job of helping you visualize your overall layout, so it's essential that you generate your own diagram, even if it's only on paper. It's important that you include the layout of the underlying storage if you're using some type of hardware RAID device.

The vxprint utility is usually your best source for storage layout information, although it takes time to understand what it means. You should also consider using a tool like "save-vxlayout" to regularly get a copy of this off of the local system for disaster recovery purposes. If you're using EMC storage arrays, you might also consider using emcprint, a script which adds physical disk information to the vxprint output.

Collect Performance Statistics

It's also vital to understand your storage performance characteristics before making significant changes. Again, you ideally want to include performance statistics from underlying storage. All of you should be familiar with the standard Solaris disk performance utility, iostat. Most of you have probably also seen vxstat, which is included with VERITAS Volume Manager and allows you to get similar

performance numbers about logical objects managed by Volume Manager. VERITAS has other performance analysis tools available internally that you may be able to convince them to share, including vxfsar, which provides extensive VERITAS File System (VxFS) performance statistics.

However, when you're using hardware RAID, these commands don't provide any insight into how the underlying components are behaving. For instance, you really need to know when cache hit rates start to fall off, particularly with write cache. With EMC Symmetrix arrays, you need to install the additional package called symcli. With Clariion products, use navicli getcache. You may need to run navicli setstats -on if you see that statistics are not enabled.

Presenting these statistics meaningfully is more challenging, but a couple of tools make this process easier. Cricket is a good way of doing trends-based monitoring of more than just network equipment. We've found it useful for tracking disk performance, filesystem usage, network utilization, etc. By allowing you to create views of all of these statistics together, Cricket makes it much easier to see when you are starting to hit a bottleneck in one of them. However, when all you want to do is collect some quick performance statistics from Volume Manager, you might give vxstat2gnuplot a try.

You should also take a quick look at your backup and restore times. In growing environments, it's critical to know how long restores will take and whether this is acceptable to your management. We've found that in certain environments, incremental restores, for instance, can take many times longer than full restores.

Looking At What You've Assembled

Trends

Detailed graphs of system performance and utilization can be crucial for several purposes, with

capacity planning probably being the most obvious one. Assuming linear growth, you can easily determine when you will need to buy disks, add servers, or move users to a new machine. In other words, it shows you roughly how long you have before you reach capacity. You can sometimes also use these graphs to determine what that capacity is, once you have an example of a system that is overloaded.

These graphs can also provide an alarm system of sorts for problems you've never run across before. The most common examples of this are sudden changes in utilization, such as those shown in the graph of interface utilization (Figure 1).

Peak usage times

In order to optimize system performance and availability, you need to understand when your peak usage occurs and how extreme it is. These peaks often

occur at somewhat different times than your management may believe, so be sure to collect your own data to see for yourself, possibly using *cricket* or *vxstat2gnuplot*, as mentioned above.

In many cases, you may find that you have multiple peaks, possibly corresponding to users in different time zones. With careful planning, it's possible to use your knowledge of these peaks to share resources more effectively. In the example above (Figure 2), you can see how sharing storage more effectively between users with different usage patterns might allow you to more fully utilize your storage.

Of course, sharing resources with users around the world can also reduce off-peak periods, which you may have relied on for maintenance or to improve backup performance.

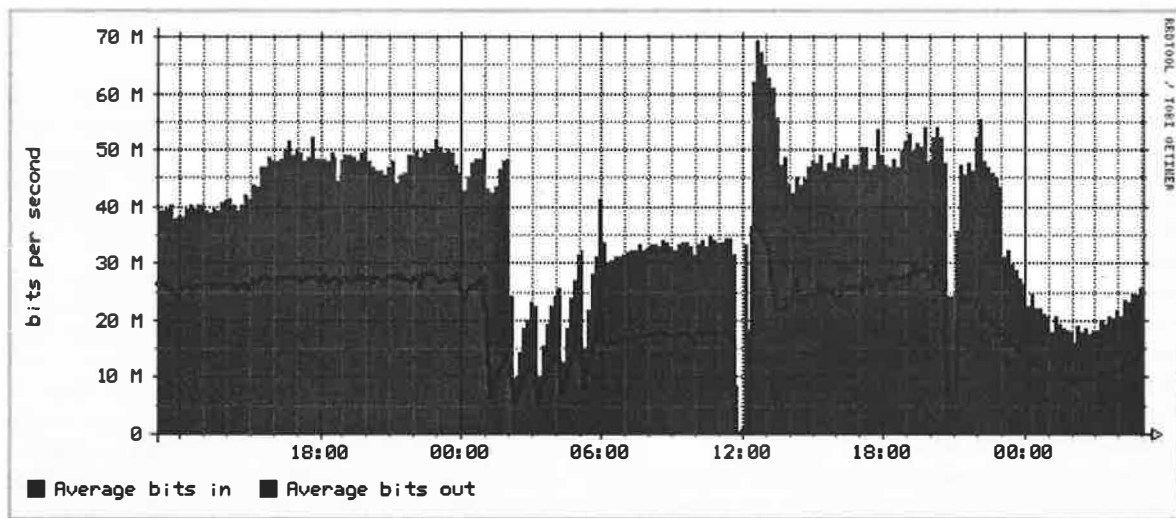


Figure 1: Example of Cricket data used to understand anomalous trends.

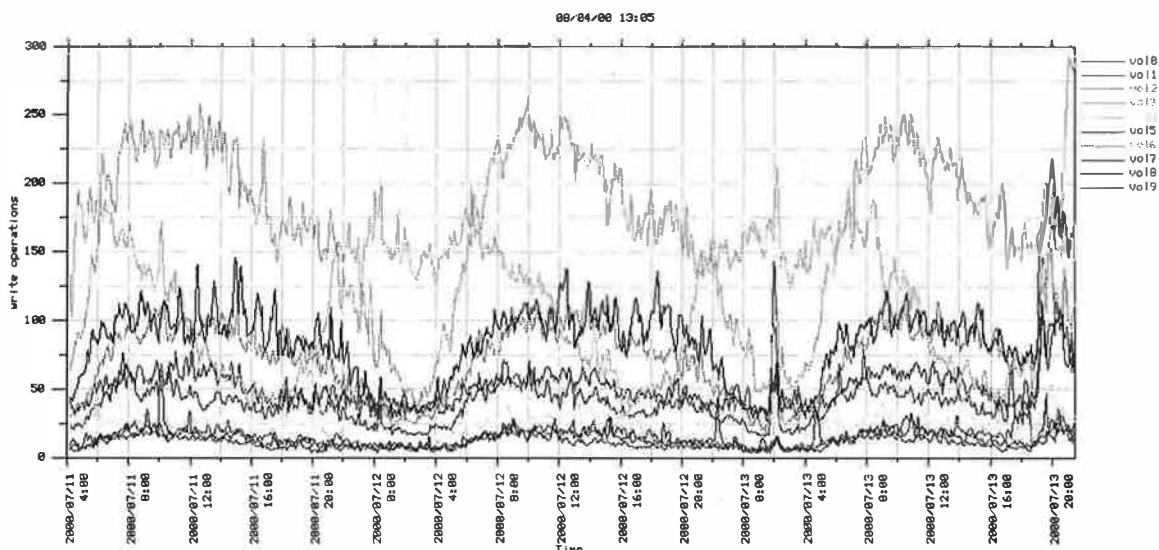


Figure 2: Example of *vxstat2gnuplot* graph.

Hardware RAID Write Cache Hit Rate \neq 100%

This typically means that the write cache is overflowing. In other words, the physical disks are not able to keep up with the data being written to them, and even an extremely large write cache will eventually fill. Under these circumstances, write service times (average write times in vxstat) can quickly increase to over ten times their normal value.

Adding additional cache is usually of limited benefit, since the writes still must be committed to disk, and will eventually fill it. The best approach is to spread busy volumes across more disks, which on some systems might prevent you from using the full capacity of each drive. Upgrading to faster drives can also be of some benefit.

Disk Read/Write Times Frequently Exceed 100ms

It's hard to provide exact thresholds that indicate disk performance problems, however access times in excess of 100ms almost always indicate that something is wrong. One thing to look for is a few disks with significantly higher access times than most others on a system. In some cases, access times as low as 20ms may be a sign of trouble.

Reducing The Impact of the More Troublesome Single Points of Failure

Careful storage management and hardware selection can eliminate most of the causes of long sustained outages, but OS, application, non-redundant hardware, and mischievous users remain as sources of occasional system crashes and downtime. The following frequently result from these situations and, in some cases, lead to each other.

Filesystem corruption

When filesystems are not unmounted cleanly, it is required that they be checked for consistency. This process usually occurs very rapidly with VxFS filesystems because all pending changes have been tracked via intent logs, and thus, only those changes need to be validated. Anything that does not appear in the intent logs is assumed to be consistent. With VxFS filesystems, there are four states that the filesystem can be in during a check.

1. The clean bit is set, so no check is required.
2. The log replay is successful, so the filesystem is marked clean.
3. The log replay is unsuccessful, requiring a full check.
4. The log replay is successful, but corruption is later detected.

When a full filesystem check is necessary, as in case three, it can be extremely time consuming. We've seen this take as long as 24 hours on a 500 gigabyte filesystem containing over 50 million files. The only way to reduce the time a full filesystem check requires is to create smaller filesystems. You should determine the maximum acceptable time your filesystems can be

down in the event of a full check and size them appropriately. If you have multiple filesystems that require a full check, be sure to do them in parallel to reduce the duration of the outage.

As for the fourth case, it should never occur. However, if it does, you will find out what a metasave is – hopefully you never will.

Volume Synchronization

Volume Manager's dirty region logging (DRL) provides logging features similar to those provided by filesystem intent logs. DRL is intent logs on the volume block level. It protects you against the need to resynchronize mirrored or RAID-5 plexes after a shutdown where the volume could not be marked clean. As with filesystem intent logs, there is still the chance that a full resynchronization is required, but it is possible to decrease the time and performance impact of this.

Volume resynchronization can take many times longer to complete on an active volume. Consider waiting for resynchronization to complete, or nearly complete before restarting services. You can see the progress of Volume Manager tasks, like resynchronizations, with the vxtask command in version 3.x. With version 2.x, you can watch resynchronization progress with vxstat -f b that will show atomic copies, which are copies between plexes.

When a mirror or a volume snapshot is being attached to an already mirrored volume, you can reduce the performance impact by setting a preferred plex. By telling Volume Manager to prefer one plex over another, it will perform all read operations from the preferred plex only. This is important because when a third plex is being attached to a volume, it will synchronize from only one of the existing plexes. If you force all other reads to occur from the remaining plex, overall performance of the volume should improve. You can determine which existing plex is being used to synchronize the new plex by using vxstat -f b and look at which disks are performing the atomic reads.

Damaged Data Due To User or Application Error

Even if you manage to protect against corruption due to system crashes, you still have to contend with damage caused by rogue users and applications. The traditional approach to protecting against this relies on regular backups. Unfortunately, restoring from backup usually means that you lose up to a day's worth of changes, and these restores can often take a long time to complete. A couple of VERITAS technologies allow you to recover from this type of failure much more quickly: Volume Manager snapshots, VxFS snapshots, and VxFS checkpoints.

A Volume Manager snapshot is essentially a write-only mirror of the volume that is broken off and becomes its own volume. The only performance impact on the original volume is during the initial

synchronization. Another disadvantage is that it requires as much disk space as a full mirror of the original volume. Since it becomes its own volume, it assumes all of the characteristics of a normal volume including persistence across reboots. However, it is created with the `nolog` option set, so you will need to stop the volume and change this parameter before you attempt to associate a logging plex. At any time, you can mount this in place of the original filesystem to revert all changes or mount it and copy individual files that need to be restored. A Volume Manager snapshot is a great way to take a backup of your data with minimal impact to the primary filesystem.

A VxFS snapshot is mounted by associating storage with a mounted VxFS filesystem. It reflects the state of the original VxFS filesystem (known as the snapped filesystem) at the time the snapshot was created. As the snapped filesystem changes, the original version of those changed blocks gets copied to the snapshot. The main advantage of a VxFS snapshot is that it uses less disk space than a Volume Manager snapshot, only requiring enough storage to accommodate the blocks that change during the time the snapshot is mounted. Unfortunately, it can be hard to accurately predict how much disk space those changes will require – the snapshot becomes disabled when the underlying storage fills up. Filesystem performance can be significantly impacted while the snapshot is mounted, since three operations are required for each change. Also, since a snapshot can only be associated with a mounted filesystem, it is not persistent across reboots. For these reasons, it is typically only used as a way of providing a consistent image during a backup, but they don't provide a mechanism for reverting the entire filesystem.

VxFS checkpoints are a sophisticated way of providing the Volume Manager snapshot capability of reverting a filesystem back to a specific point in time nearly instantaneously. They work in much the same way a VxFS snapshot does, but they are not mounted and they use the primary storage to store the changed blocks. They are useful for performing backups using the Block Level Incremental Backup product. Since you can't mount a VxFS checkpoint, you can't easily restore an individual file.

Failed Components

Volume manager does an excellent job of providing details when errors occur. It is your job to make sure those notifications go to the right people. `Vxnotify` is used to provide email notification of volume manager errors. It defaults to sending those notifications to root on the local machine, but you can specify any email address to receive these notifications in the startup script, `/etc/init.d/vxvm-recover`, by changing the argument it passes to `vxrelocd`. Volume Manager also reports errors to `syslog`, providing another mechanism for failure notification. `syslog` is the only

mechanism by which VxFS reports errors, so it is critical to monitor `syslog` in some fashion, such as with `Netcool`.

Full Filesystems

Full VxFS filesystems can be even more troublesome than UFS filesystems. In both cases, it is possible to grow the filesystem online. However, VxFS filesystems need space for additional structural information during filesystem growth operations, making it impossible to grow a completely full filesystem.

To prevent this situation from occurring, always monitor your filesystems carefully. If you aren't completely confident in your monitoring system, you should at least create a 10mb placeholder file so that you can easily free up just enough space to grow the filesystem. Growing a filesystem by a large amount requires more space for structural information than growing it by a small amount, so it's sometimes necessary to grow it in multiple steps. However, we've been warned by folks at VERITAS that it's safer to limit the number of growth operations.

Failover

Any application or service that maintains unique, persistent data requires some sort of recovery strategy. Depending on business needs, it may be acceptable to sustain a few unplanned 30-60 minute outages per year, with a chance of a multiple hour outage. If this is not acceptable, some sort of failover scheme is necessary. Automated failover becomes even more critical if you're committed to providing such a high level of service availability that even occasional planned downtime is otherwise impossible.

A failover strategy can be as simple as having an idle system connected to the same disk array. This sort of primitive approach requires thorough rehearsal and process documentation, but it can be nearly as effective as more costly solutions.

A step up from this is what amounts to home-grown automation, or partial automation. A fairly simple script can be used to ping another machine that's running a service, and if it's not there, start that service locally. However, it can be tricky to accurately detect every possible failure mode, and the failover of persistent storage is hard to automate safely.

VERITAS Cluster Server is one of the more popular commercial solutions for automating failovers. This may be the easiest, most reliable solution if you're running one of the applications or services for which they provide a bundled solution, such as Oracle or NFS. It automates things so your failover is consistent. It understands dependencies between services to ensure consistent adherence to failover policies. It parallelizes many activities, such as checking and mounting filesystems, which results in rapid startup times. Finally, it has sophisticated safeguards to protect against what's generally called split-brain

syndrome. Split-brain occurs when two hosts try to write to the same disk, often resulting in corruption.

Characteristics of Environments Likely to Experience Extreme Scalability Problems and What You Should Be Fixing Now

Lots of Small Files

Metadata updates can be bottleneck

The VERITAS File System stores all of its metadata at the beginning of the volume, which includes things like directory information and the intent log. Certain types of filesystems, particularly those with a large number of small files, can tend to be performance bound by the speed of the disk(s) on which the first portion of the filesystem is located. The performance of intent log writes is reduced because the filesystem is also performing other operations. In some cases, this can be improved considerably by moving the intent logs onto dedicated storage, which is what the QuickLog product allows you to do. It does add considerable complexity, particularly in clustered environments, but the performance gain is often worth the hassle.

In some cases, vxstat may indicate that even after the use of QuickLog, the disk(s) that contain the first portion of the filesystem are having a hard time keeping up. Intent logs aren't the only type of metadata normally stored at the beginning of a VxFS filesystem,

but they are the only type that can be moved elsewhere. As is true in most other cases where a volume is performing poorly, the only way of decreasing the impact of the remaining metadata is to increase the number of columns in the plex (stripe it across more disks).

The simplest approach to increasing the number of columns is to attach a new plex that contains more columns (disks) than the original plex, then remove the original plex(es). If you are running VxVM 3.0 or above, you can also use online relayout, which can get you to the same state without requiring quite as much disk space along the way. Both of these methods can be accomplished with vxassist.

Backups Are Hard, Restores Are Even Harder

The only practical way to back up a filesystem with a large number of files is via the raw device, since backing up each individual file sequentially would take a very long time. However, it's critical that you have a consistent version of the filesystem to work with for the duration of a backup. Snapshots, either at the volume or filesystem level, are generally the best approach. The FlashBackup extension to NetBackup provides an easy way of automating backups via filesystem snapshots. It also has the advantage of working with both VxFS and UFS filesystems.

Restores of millions of files also need to be done through the raw device, if at all possible, since they

		Target					
Controller		0	1	2	3	4	5
	0	rootdisk1	orasrc,apps archive-01		LOG1		
	2	rootdisk2	orasrc,apps archive-02		SPARE		
	3	oradb02-03	oradb02-03	oradb02-03	oradb01-03		
	4	oradb03-01	oradb03-01	oradb03-01	LOG2		
	5	oradb04-02	oradb04-02	oradb04-02	SPARE		
	6	oradb02-01	oradb02-01	oradb02-01	oradb04-03	oradb04-03	oradb04-03
	7	oradb03-02	oradb03-02	oradb03-02	oradb01-01		
	8	oradb02-02	oradb02-02	oradb02-02	oradb04-01	oradb04-01	oradb04-01
	9	oradb03-03	oradb03-03	oradb03-03	oradb01-02		

Figure 3: Example disk re-layout plan.

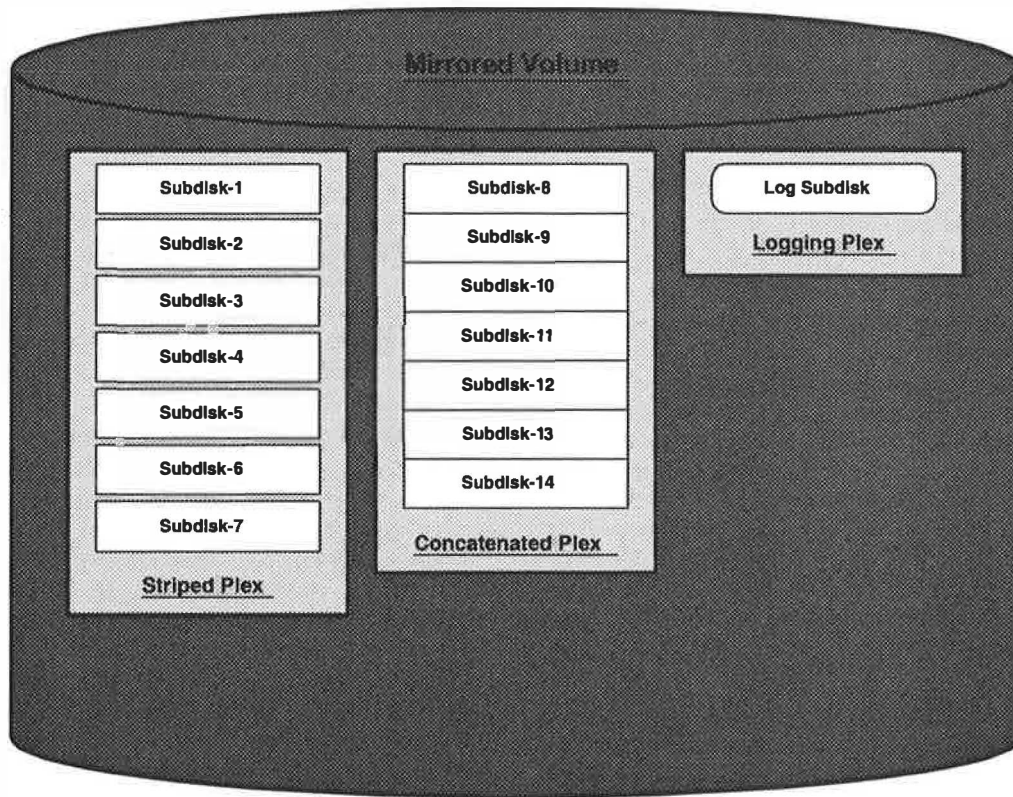


Figure 4: Example of mirrored volume with striped, concatenated, and logging plexes.

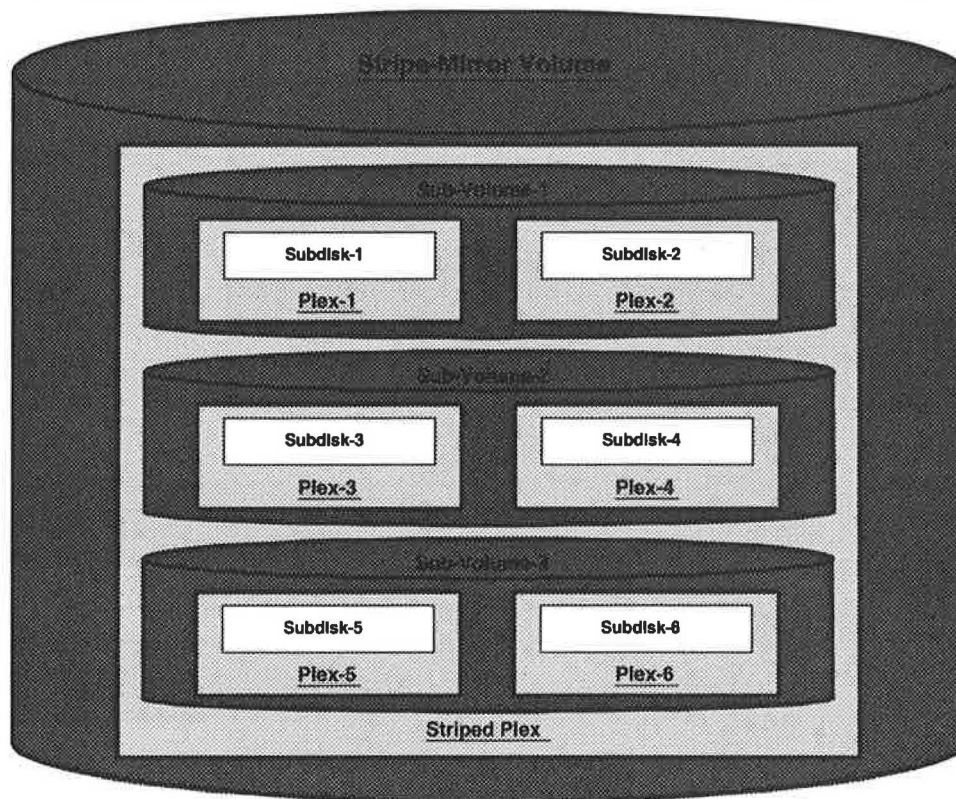


Figure 5: Example of layered volume with sub-volumes. This is also known as a striped-pro volume.

experience similar performance problems to backups. Even though incremental backups are possible and relatively fast with FlashBackup, incremental restores are another matter. Incremental restores are done through the filesystem instead of through the raw device, as other restore utilities like vxrestore and ufsrestore do. This can literally take weeks to complete when you have tens of millions of files. If you have that many files, you're probably better off only doing full backups, since they restore directly to the raw device. We've even seen full restores with FlashBackup take less time than the original backup since the restore doesn't require the inode map.

High Transaction Rate Systems

Extremely Busy Disks

From the info provided by vxstat or other tools, you'll probably find that certain disks are being used in too many "hot" volumes, or are otherwise unable to keep up with the I/O activity. If there are only a few busy volumes on the system, it's often possible to have only a portion of each physical disk in a busy volume and use the rest in more idle volumes. This allows you to increase the number of columns without having to add a lot of extra storage.

However, if almost every volume is extremely busy, you may not be able to fully utilize your disks. This is often a problem on systems with large disks, where the number of transactions per second is more than these disks can possibly keep up with. When possible, try to use small, high performance disks for your busiest volumes. If you're stuck with larger drives, consider preallocating up to 50% of each drive for a scratch volume, ensuring that other administrators don't accidentally over-commit them.

General relayout

The importance of carefully planning storage layouts before a system goes live can't be stressed enough. However, even after a system has been deployed, it's often possible to reorganize things. Start by creating a storage layout grid, similar to Figure 3.

Even if you only have one free disk available, it should be possible to gradually move data until it looks like your ideal storage plan. Online relayout may be your only choice if you don't have many free disks. It should be possible to make all of these changes without any service interruption, although the process of moving data can have a serious performance impact.

Glossary

Subdisk – A consecutive portion of a physical disk analogous to a partition. One or more subdisks can be grouped to form a plex.

Column – Groups of one or more subdisks that constitute a single stripe within a plex. For example, a 4-way striped volume would have four columns.

Plex – Typically, a set of subdisks that contain a complete copy of the data on a volume. Logging plexes are another type of plex which stores Dirty Region Logging data.

Volume – A collection of one or more plexes which make up a logical disk. It can hold a filesystem, swap device, or a raw device for a database.

Sub-volume – Starting with VxVM 3.x, you have the ability to create layered volumes where, for example, you can stripe across a set of mirrored subvolumes.

References

VERITAS File System System Administrator's Guide Release 3.3.2, VERITAS Software Corporation, May 1999.

VERITAS Volume Manager 3.1 Administrator's Guide, VERITAS Software Corporation, July 2000.

VERITAS NetBackup FlashBackup 3.4 System Administrator's Guide Unix, VERITAS Software Corporation, June 2000.

VERITAS User mailing lists, maintained by Doug Hughes: <http://mailman.eng.auburn.edu/mailman/listinfo/>.

Repository of helpful storage management hints, maintained by Doug Hughes: <http://www.eng.auburn.edu/pub/mail-lists/ssastuff/>.

Appendix: Third-party tools

Cricket

This tool is written and maintained by Jeff Allen. It can be downloaded from <http://cricket.sourceforge.com>.

emcprint

The emcprint script produces It prints the director controller LUN for each disk that has an EMC name. This assumes you have run the createMatrix script (depends on symdev, included with EMC's symcli tools, a separately licensed product). This and the following three utilities are available from <http://www.vxideas.org>.

save-vxlayout

This tools preserves critical Volume Manager layout information and gives you a fighting chance of recovery if your configuration database gets corrupted. This sort of corruption can be caused by a split-brain scenario, where two systems try to update a disk group at the same time. It can either periodically write this backup info locally (to a file on a non-shared disk group) or email it off of the machine. We've also found it very useful for remote troubleshooting to use this to collect all of this info on a central management machine.

vxstat2gnuplot

This tool generates graphs of performance statistics of various VxVM-managed objects. This also depends on running rotatevxstat (which is also useful for determining bandwidth requirements for SRVM).

drive-status

This is a simple script which, when run from cron, notifies you of downed tape drives in NetBackup. This seems like something that NetBackup should provide natively, but at least as of version 3.2, it missed this. In large NetBackup environments, downed tape drives are often the leading cause of failed backups.

Author Information

Karl Larson attended Harvey Mudd College, majoring in Biology. He accepted a job as a Systems Engineer for WebTV Networks in 1996, where he helped support their production environment and scale it to support nearly a million users. He left to work at GNAC starting in 1999, where he was a Systems Architect, among other roles, helping design highly available customer environments. He has recently started work at Tellme Networks as a Senior Systems Engineer.

Todd Stansell attended UC Davis, majoring in Computer Science Engineering. He has been an employee of GNAC since early 1998, supporting customers such as WebTV, Exelixis, and Redherring. At GNAC, both Todd and Karl have worked with VERITAS Consulting to support some of their most critical customers.

User-Centric Account Management and Heterogeneous Password Changing

Doug Hughes – Auburn University

ABSTRACT

There are a plethora of password changing programs available for users and systems administrators to use. Most of the existing password changers, however, fall short in some way. Either they are still text based, requiring users to use unfamiliar tools in unfamiliar environments, or they are part of an all encompassing framework – designed to completely supplant the entire existing account creation, maintenance and distribution process. Our program is designed, first, to be usable without training or human support. Second, it is designed to use existing distribution databases such as NIS and LDAP. We integrate many of the features of prior password changers such as the Cracklib library, character classes, and rules for password selection. We try to provide easy extensibility both in terms of database support and by providing other user-focussed programs that use the same authentication framework.

Introduction

Auburn University College of Engineering has two primary platforms for supporting a broad clientele. These are Windows boxes which use SMB [1] for file service, and UNIX boxes which use NFS for file service. In order to use any network services, a user must be authenticated. For the UNIX side, this means NIS, NIS+, or LDAP.

To provide authentication to NT boxes we use Samba [2]. To do this semi-securely [3] requires a separate encrypted password database. In their respective, encrypted formats, the UNIX password and the Microsoft password are incompatible, irreversible one way hashes; there is no method to generate one from the other. This means neither system's builtin password changing mechanisms can be used. This irreversibility problem can be applied by extension to other one way password formats such as BSD44, Apple formats (e.g., CAP [4]) and several Linux formats.

Early in 1999 we decided that we should replace the existing – functional but archaic – text-based password changer with a Web-based mechanism. The old system required users to login to a UNIX machine to change their passwords. Many of our new users had no experience with applications like telnet, ssh, or even UNIX. Text based solutions requiring login to an unfamiliar environment were difficult to support. A Web design allowed greater platform availability and improved usability over our previous system. We also had an opportunity to revise the entire password changing system and integrate other services such as email forwarding.

We wrote our code primarily in PHP [5]. Some functions required a PHP loadable module written in C. One helper program was written in Perl.

Prior Art

A lot of work has been put into account management systems over the years. Many USENIX, LISA,

and even */login:* articles have been written concerning these works. Nearly all of them have a systems administrator focus, which is natural when one considers that systems administrators are always trying to make their jobs easier. However, the works tend to focus on systems administrator specific tasks such as creating, adding, deleting, and ongoing maintenance of accounts. When usability was addressed, it was usually most evident from a systems administrator's point of view: make less work; provide more scalability.

Auburn University College of Engineering (hereafter referred to as AUCOE) already had account maintenance tools in place with no obvious need for wholesale replacement. We required an intuitive user interface – to allow users to be able to do simple tasks on their accounts without any sort of training – and minimal support.

The main differences between the AUCOE framework and prior implementations are:

1. It integrates prior work and tools such as Cracklib [6], character classes, etcetera. (some, but not all, of the others also do this.)
2. It provides easy extensibility to other formats such as LDAP and custom databases.
3. It is much simpler in scope than the others. Some of them are enormous and handle everything from account creation to interfacing with a human resources database. The AUCOE changer provides a means for users to easily change their passwords.
4. Our main focus is user usability vs. administrator usability.
5. The incremental approach builds on top of existing frameworks such as NIS, NIS+, Samba, and flat files, rather than replacing them.
6. Decoupling password choosing from distribution and propagation allows us to not have to worry as much about locking and conflict resolution.

The author was not able to gather meaningful data – because of age and lack of access to the original papers – on these password and account management systems: Maryland [7], ACMaint [8], Apollo [9]. References are provided at the end for those having hardcopy proceedings.

Many of the features of the following command-line, pro-active password checkers have been integrated into our password changer: npasswd [10], passwd+ [11], ANLpasswd [12], Epasswd [13]. The password vetting routines from these programs tend to be tightly coupled with an existing command-line interface and propagation mechanism. Rather than extracting the proactive changing logic and rules from any of these programs, it was easier for us to use Cracklib and construct the various password rules and classes in native PHP.

Information about current availability of and updates to other user account management systems was difficult to collect. Some of the systems listed may have had updates more recent than the author was aware. Some of them were designed to manage access control where certain users are only authorized to use certain machines, or the home directory may differ based upon the machine. All of them had, as a primary focus, the goals of modernizing and automating the maintenance of user accounts at a site; password changing, password synchronization, and usability, if mentioned at all, were typically secondary. The most currently relevant systems are compared in Table 1 based upon the following criteria:

- Design philosophy
- Custom database vs. existing database (e.g., NIS, LDAP)
- Distribution and synchronization methods for accounts and passwords
- Extensibility – perceived degree of difficulty of adding a new output format, heterogeneous machines, configuration intricacies, etc.
- User interface focus
- Language(s) written using and/or configured with
- Release status (may be out of date)
- Other characteristics that may be of interest

Genesis

The framework used by the AUCOE password changing system came into being ad hoc. The project that originated the framework provided a means by which users could forward their email. The previous forwarding program was text-based and required users to login to a UNIX machine to run it. The number of users wishing to forward their email was growing. The time spent supporting these non-UNIX users was growing proportionately.

We desired to avoid CGI and its inherent security difficulties and call-out overhead. After a short period of investigation, comparing the tradeoffs of mod_perl [19] versus learning PHP, a new language, the author chose PHP. PHP combined C and Perl syntax without the special variable cruft. PHP had support for persistent file handles. Finally, PHP appeared to be easier for non-experts to learn and use quickly.

Name	Design Philosophy	Database Type	Distrib./ Sync.	Extensibility
Shuse [14]	sync accts across mult mach's with central database	custom, central	NFS, NIS, FTP, sockets	uses expect
Agus [15]	acct. over multi archs using CCSO [13]	custom (fixed?)	Kerberos, VMS, Unix, etc.	fixed DB but godo for new accounts
NAMS [16]	client/server daemon with modules replaces NIS	custom but open (ASCII key & data)	TCP/IP socket (client/server)	modular design
Accountworks [17]	ease hiring process and account creation	Sybase (other..)	central client/server but no passwd sync	probably difficult
Ganymede [18]	provide central DB push to NIS, LDAP, etc.	Central, RAM, OO	NIS, NIS+, LDAP, etc	properly changing schemas not for timid
Auburn COE	Provide usability atop existing dist. mechanisms	use existing (NIS, etc.)	use existing (NIS, etc)	Use PHP, module, or other (e.g., TCP/IP)

Table 1: Survey of similar systems.

In keeping with the tenets of user-interface design, the password change form was designed to be simple but usable. User feedback was incorporated to make the form what it is today (Figure 1). The results were positive both in terms of user satisfaction and reduced support by administrative staff. We were encouraged to try again: with passwords.

Design Goals

Our system for changing passwords was designed to meet certain goals:

- It must have universal accessibility.
- It must be secure.
- It must be easy for novices to use.
- It must not rely upon extensive online help.
- It should *have access* to online help for users having trouble.
- It must not inhibit expert users.
- It must be scalable.
- It must provide clear and meaningful, jargon-free responses.
- It should build upon existing distribution and synchronization mechanisms: NIS, NIS+, Samba, LDAP.
- It should be easy to extend as other distribution and synchronization mechanisms become available.
- It should borrow techniques from prior password changing implementations.

We now provide finer detail on these goals. Why are they here? How are they achieved?

It Must Have Universal Accessibility

In other words, use the Web. It is time consuming to develop clients for various architectures. The flexibility and speed of dedicated, platform-dependent password clients was not indicated.

It Must Be Secure

Because users are providing information that gives access to their accounts, and because secure, local access cannot be assumed (though it could be enforced), it is imperative that *no* information be transmitted over the network in the clear. There are multiple paths of transmission involved. See the **Security** section for more details.

The Web server must verify the username and old password for authenticity. Our implementation provides multiple ways to do this.

1. Make the Web server a NIS slave.

PHP can directly access DBM files, so verifying passwords by giving the Web user account read-access to the encrypted password database is possible. (Before anybody gets apoplectic, please continue reading.) If your Web server is single-purpose, contains no extra CGI, no other user scripts, and no unknown functionality, this is relatively safe.

2. Use the provided pwcheck daemon.

The pwcheck daemon uses a secure algorithm to verify the username and password. If neither is correct, a non-specific negative result is returned: users are informed that either their username or password is incorrect.

Name	User Interface	Construct/ Config	Release Status	Other Features
Shuse [14]	text based	Tcl/Expect with a little bit of C	Only Sheridan College	
Agus [15]	unknown	90% Perl, 10% C	N/A at publication	account clusters, fine-grained machine access control
NAMS [16]	text based (customized npasswd)	C daemons, ASCII DB config	prev. ftp.cs.jmu.edu	account clusters
Accountworks [17]	Web based	DB driven + Perl, sybperl, Notes, sh, etc.	not available	designed for non-techies; huge scope
Ganymede [18]	Java Applets, count on training	140K+ java lines, web-based config	www.arlut.utexas.edu/gash2	DB limits, good access control
Auburn COE	Web forms	PHP, some C	www.eng.auburn.edu/~doug	easy to add new features, e.g., forward mail

Table 1b: Survey of similar systems, cont.

3. Use another authentication mechanism.

While not provided, it would be fairly easy to interface with LDAP, NIS+, or another authentication database to verify a user's identity. One of the difficulties of this approach is choosing between the lesser of two evils. Do you wish to give your Web user root access to the entire authentication database or do you wish to provide a *setuid* root helper program that becomes the user prior to checking the encrypted password? This is the primary reason why the *pwcheck* mechanism is provided. It is discussed in more detail in the **Security** section.

It Must:

Be Easy For Novices To Use

Rely Upon Extensive Online Help, and

Provide Access To Online Help For Users Having Trouble

These three goals are complementary, and might even be merged into one. The distinctions

among ease of use, having online help, and relying upon online help, though subtle, are important. The interface must be intuitive. One of the well-known facts of user-interface design is that users rarely, if ever, read any online help before attempting to do something. They usually dive in oblivious to the circling sharks. The interface **must** be intuitive.

If a user cannot sit down and use a password changing program without training or referring to a manual, the program should be re-evaluated. We have gone through several iterations of user feedback and modification. There is, however, a difference between using the program without requiring a manual and referring to supplemental help should the password choice be insufficient. We avoid the requirement to read online help before-hand by providing abbreviated rules for choosing a password right at the top of the Web form. The rules (Figure 2) are specific enough to be easily understood, but small enough to fit entirely on the form and in the browser window.



Your password was rejected because it is based on a dictionary word. Please choose a better one or [follow this link](#) for advice.

Change CoE Password Form

- Passwords must have six(6) to eight(8) characters
- must contain at least one(1) uppercase letter
- must contain at least one(1) lowercase letter
- must contain at least one(1) other character
e.g. 0 1 2 3 4 5 6 7 8 9 ! @ # \$ % ^ & * - + = \ | > < ? / - ' " ' "
- Your password cannot be a word found in any dictionary! (English, Foreign, or other)

For hints and clues about choosing a good password that won't keep getting rejected, [follow this link](#).

Username:	<input type="text" value="jdoug"/>	Please enter your CoE username.
Old Password:	<input type="password" value="*****"/>	Please enter your old CoE Password.
New Password:	<input type="password" value="*****"/>	Please enter your new CoE Password.
New Password(again):	<input type="password" value="*****"/>	Please enter you new password again.
<input type="button" value="Change Password"/> <input type="button" value="Reset"/>		

[\[Eng. Home \]](#)
[\[Search \]](#)
[\[Table of Contents \]](#)
[\[Feedback \]](#)
[\[Address Book \]](#)
[\[Help \]](#)
[\[OASIS \]](#)
[\[LAU Directory \]](#)
[\[LAU Technology Hotline \]](#)
[\[LAU Search \]](#)



Figure 1: Sample user interface.

After each rejection of a password based upon the given rules, the user is given the opportunity to follow a link which gives suggestions for and examples of choosing a password. Failure messages are communicated in bold red letters in a larger font and written at the top of the page as the form is redrawn, giving clear visual feedback about problems.

It Must Not Inhibit Experts

By providing a short set of rules, but not overwhelming the user with extensive directions, the interface is usable by experts.

It Must Be Scalable

Most of the scalability (and locking) issues are pushed back to the distribution mechanism and thus avoided. The interface must still, however, be able to handle the case where a number of users connect simultaneously. The worst case scenario can probably be attributed to university environments when new accounts are generated at the beginning of each term. Even in this scenario, however, it is extremely unlikely for more than a few users to change their passwords at a given time. Many of our users opt to keep the random FIPS-181 [20] style passwords that they are issued. Even if there was a Freshman computer lab where the instructor dictated that all students must change their passwords, changes would still not be simultaneous. People read, type, and think about their passwords at different rates. The

worst case would probably not be more than 5 to 10 simultaneous password changes, which is easily achieved even with a modest PC as the Web server. In practice, even in its 18 months of existence, it has been unusual to have more than one person trying to change a password in any given five minute interval.

Locking issues, similarly, are not a great concern. Most of the methods to change passwords in the existing back-end databases are already serialized. NIS, NIS+, and LDAP already have their own locking mechanisms. Even when we update our Samba password file with the encrypted MD5 hashes, we use a simple, Perl, single-threaded daemon to handle network requests from the Web server and update the flat Samba password file. (See the **Extensibility** section).

It Must Provide Clear and Meaningful, Jargon-free Feedback

User feedback issues have been partially previously addressed. At the risk of appearing to climb onto a soapbox, this is an area that is ignored far too often by systems administrators developing user interfaces for users. Users do not care – and should not be burdened – with messages about errors in some anonymous line of computer code. We systems administrators, as a community, are particularly guilty of these transgressions. The AUCOE program gives messages in plain English such as ‘New

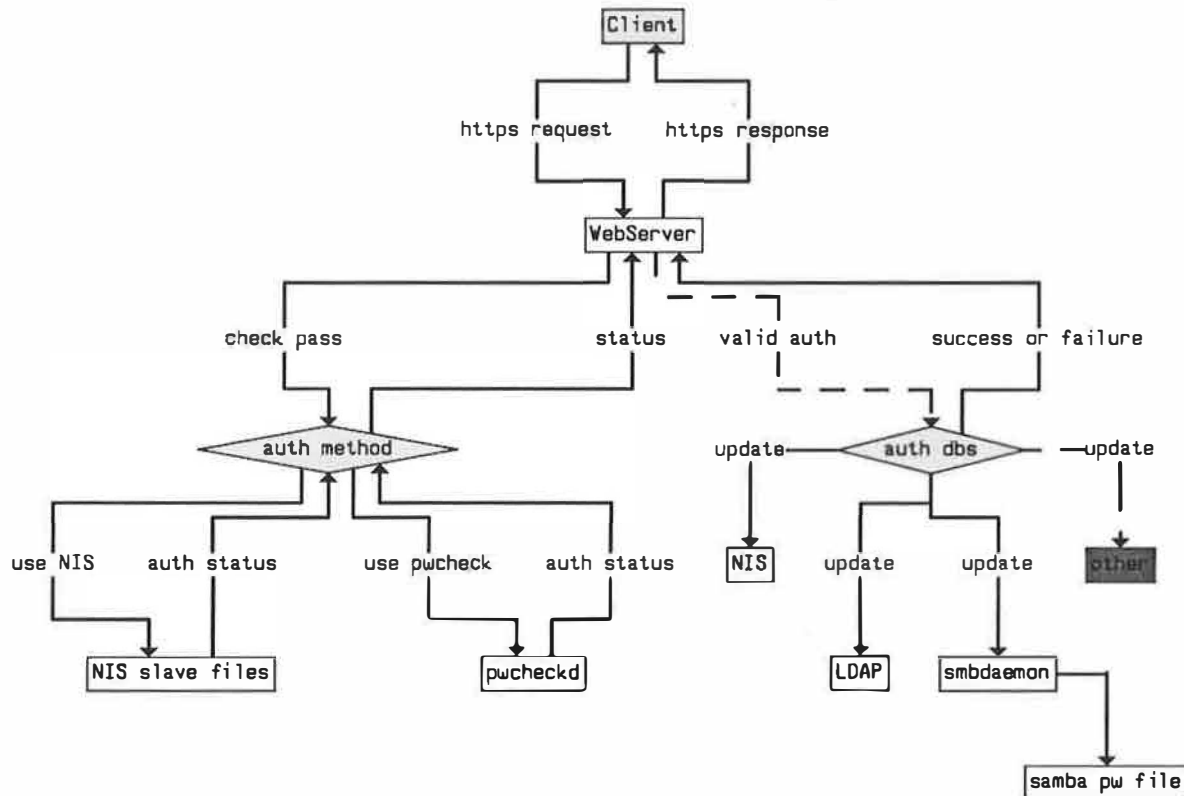


Figure 2: Password verification and update flowchart.

passwords do not match. Please retype them.', or 'Passwords require at least six characters', or 'The new password you entered is too similar to your full name or username.' Upon successful completion, the program explicitly tells the user: 'Your password has been changed successfully and will be ready for you to use anywhere in engineering in the next 10 minutes.', giving them explicit expectations and scope.

It Should Build Upon Existing Distribution And Synchronization Mechanisms

One of the drawbacks – or features, depending upon your point of view – with some of the other systems is that they require wholesale replacement of existing mechanisms. This means modifying

every machine, or replacing the entire account generation mechanism, or tailoring lots of configuration files. Sites starting with nothing, sites having special access requirements, or sites mired in complicated legacy scripts may have no qualms about replacing an existing system. We provide an incremental approach to sites that, like us, already have established account generation and maintenance mechanisms. It is not always possible to build incrementally, but in our case it was desirable.

It Should Be Easy To Extend as Other Distribution and Synchronization Mechanisms Become Available.

PHP, like Perl, is a "kitchen sink" language. It contains builtin interfaces to many databases and a

```
// Who do you want administrative mail (errors, etc) send to?
$Admin_Staff = "admin";

// General Vars - length of passwords
$MAXlength = 8;
$MINlength = 6;

// Error logging (daemon|error) (consult syslog.h)
$SYSLEVEL = 27;
// comment this out if you don't have NIS.. If you do this, you
// better have LDAP (or skills to modify passwd.php3)
// Note - this is for submitting the password change.
$have_NIS = 1;
$NIS_domain = "eng.auburn.edu";

// This is for checking the current password
// For machines with direct access to shadow password file,
// set $direct_access = 1; and set password file like the following
// line:
// $pw_dbm = "/var/yp/domainname/passwd.adjunct.byname";
$direct_access = 0;

// Where do you keep your user .forward files?
$forward_path = "/forwards";
// If you use the forward function, choose the location of your php_file
// helper auxiliary program
$php_forward_helper = "/etc/local/php_file";

// For getting page count database info. Basically, nobody really
// cares about this but us (Auburn). It's used in paper.php3.
$getpages = "/etc/local/getpages";

// Change these for LDAP
// ldap_uid should be an account with create and modify privileges on
// everything in your Samba and NIS ldap trees.
$have_ldap = 1;
$ldap_server = "ldap.your.domain";
$ldap_uid = "root";
$ldap_pw = "rootpw";
$ldap_smbdn = "ou=people,dc=eng,dc=auburn,dc=edu"; // Samba tree
$ldap_nisdn = "ou=people,dc=eng,dc=auburn,dc=edu"; // NIS/UNIX tree

// For socket based manual updates to an smbpasswd file, this is
// the host. Uncomment it here and change it as appropriate
// $SMBHOST = "smbhostname";
```

Figure 3: PHP configuration file.

rich set of string processing and system interface functions. It also has a well-documented and easy to use C API for extension. The Cracklib module is written as a loadable extension, as are methods for generating Samba hashes, and calling NIS for password update. TCP/IP sockets and cryptographic functions are also builtin providing yet another means for extension: client/server communication.

Architecture and Implementation

The code, where possible, was written in PHP for Apache [21]. A dynamically loadable PHP module was constructed as an enhancement. We also created a Perl helper daemon on the Samba PDC to accept network connections from the secure Web server and update the Samba password file.

Figure 2 shows a high-level flowchart of a password change transaction. The 'auth method' box encapsulates all of the functionality for checking to see that the username and old password are valid, as well as running the rule checks and Cracklib on the new password. Upon successful validation, the respective authentication databases are updated and status is returned to the user. The dashed "valid auth" line is conditional upon success of the authentication method. The dashed "update" line to the *other* box represents generic extensibility.

Configuration is currently managed through a PHP include file: *globals.php3*. Figure 3 shows a sample, commented *globals.php3*. This file lets you define and configure your authentication services so that when a user changes a password, the appropriate databases are updated.

The provided PHP loadable module is written in C. There are four main functions in this module. The *checkpass* function is used to verify the username and password as accepted from the user's browser. It calls *pwcheckd* on the server as detailed in the **Security** section. The *crack* function calls the Cracklib library with the user's desired new password. The *passwd* function calls the RPC *yppasswd* function to change the user's password in the NIS password map; the NIS master is set at compile time in the Makefile. Lastly, the *smbpasswd* function takes the user's plain text password, creates the NT hash pair, and returns the ASCII representation to PHP for transmission to the *smbdaemon* program. Transmission is accomplished with standard sockets.

Smbdaemon is a simple Perl program running on the Samba PDC machine. It is a 130 line Perl program at this writing. It listens on a well known port for updates, and then writes them to the private Samba *smbpasswd* file.

Security

There are five major areas where security needs to be addressed:

- On the Web server machine.

- Between the Web browser and the Web server.
- Between the Web server and the *pwcheckd* daemon (if used).
- Between the Web server and the authentication database (during updates).
- In ancillary programs.

On The Web Server Machine

Changing passwords has broad security implications. We strongly recommend that you dedicate a machine to these 'user services'. Do not install this software on your general Web server. Do not give users interactive accounts on the machine itself. Users given direct access to the machine may have access to files that they should not.

Browser to Server

As discussed in the goals section, the transactions between the client browser and the Web server must be encrypted using HTTPS, preferably using 128 bit clients. You should force your Web server to only accept high grade security connections.

If you decide to setup the Web server as a NIS client, you will authenticate users by directly comparing their encrypted passwords in the shadow password file with their passwords passed from the browser. In this case, you do not need to worry about passing the user's password over the network (because the files are local – except of course during periodic NIS synchronization updates). You do, however, need to be doubly certain that you do not give any users shell access to the machine. Additionally, you should setup NIS to use shadow passwords.

Pwcheckd

Pwcheckd has been designed to allow the Web server to verify the password of a user without passing the clear text or the encrypted version of the password over the network. The authentication transactions are summarized in the following list.

1. The Web server calls a function to verify the username and password which in turn calls *pwclient* in the loadable module.
2. *Pwclient* connects to *pwcheckd* on the designated port of a known server and asks for the UNIX password salt [22] for a given username.
3. *Pwcheckd* checks *tcp_wrappers* for valid client access.
4. *Pwcheckd* sends the salt back to *pwclient*.
5. *Pwclient* uses the salt and UNIX password algorithm to encrypt the user's supplied password.
6. *Pwclient* makes an MD5 hash of the user's encrypted password and current time and sends both hash and time to *pwcheckd*.
7. *Pwcheckd* receives the hash and time¹ from the user, and compares *pwclient*'s time with the

¹You should use a time synchronization program like NTP [23] between the Web server and the machine running *pwcheckd* to keep the clocks synchronized.

server's time. If the time is not within plus or minus 30 seconds, the attempt is logged as a possible replay attack and the user is not authenticated.

8. *Pwcheckd* makes an MD5 hash of the user's actual password and time supplied by *pwclient* and compares with the hash from *pwclient*. Success or failure is returned to *pwclient*.

Server to Database Updates

Server to database update security is going to be dependent upon the authentication databases used. For LDAP, the current implementation uses the LDAP root password to open the database and update the encrypted portions of the user's LDAP entry. This password is currently stored in the *globals.php3* file. Since *globals.php3* contains only PHP variable assignments, as long as you do not have your Web server setup to allow people to fetch PHP3 files, it will be safe. To bring home an earlier point: do not give users shell access on this machine.

For NIS, PHP calls the RPC *yppasswd* function (in the PHP loadable module) which exposes the user's old password in clear text while the new password is transmitted in encrypted format. Because the update is fast, the risk is small and mitigated in other ways. (You do have your Web server and NIS server on the same secured, switched network, right?)

You may be wondering why the *pwcheckd* mechanism appears to be so much more stringent and secure than the database updates. The underlying authentication databases have a considerable influence on the update mechanisms available. The author hopes that these mechanisms can be improved in the future. For now, keep your Web server and your master databases on the same, secure network – preferably locked in a room or closet with no user accessible jacks or VLANs. Even if you do not follow this advice, the exposure of an encrypted password to prying eyes is typically² less of a concern than exposure of the plain-text original.

Ancillary Programs

For the contributed Samba password update program, the NT hashed password pair is sent via a socket to a server program running on the Samba PDC which replaces any existing Samba entry with the new one. This is a generic mechanism that could easily be extended or replaced. As configured, *smbd* only accepts connections from the Web server. No plain-text password is transmitted, but the NT hashes are plain-text equivalent in that, if they are stolen, they will give access to the server as that user (a well-known Microsoft problem).

There is another ancillary program, *php_file*, not directly related to the password changing functionality. If you do not wish to use the mail forwarding

²See following paragraph.

functionality, remove *formail.php3* and the *php_file* program. Since Web servers are typically run either as the user nobody, or as a special account such as www or www-data, the helper program must be setuid to be able to edit the user's .forward file. While the *suexec* functionality of Apache would appear to be suited for this task, it has a number of shortcomings that make it less ideal.

- *Suexec* is not installed by default and requires special re-compilation of the Web server in many distributions. This, in turn, requires additional configuration, care and knowledge. Our set of tools is intended to be easier to run out of the box.
- The program to be executed by *suexec* must be resident in the WWW space and owned by the effective user doing the executing. Instituting this would require that all users have a copy of *php_file* in their Web directory space. This could be automated, of course, but would necessitate yet another process during account creation and deletion, as well as a small waste of space. This residency requirement could potentially involve thousands of new directories in the WWW space as well; since we recommend a dedicated Web server, a new directory would need to be created for every user and the *suexec*-able *php_file* must be copied into that directory.
- Altering the *suexec* code is potentially hazardous. There are many warnings about doing so. Though constructing new setuid code has its own perils, the author opted for the *perceived* simplicity of this approach. The alternative was altering a complex program covered with virtual no trespassing signs and barbed wire.

Consequently, *php_file* is setuid root. The username and password are given to *php_file* to verify. In its current implementation it lacks the authentication flexibility of the password changer. Instead, it uses the system to fetch the user's actual password given the supplied username. It then encrypts the supplied password and compares it with the actual password. If they do not match, the user is informed. If the actual password and supplied, encrypted password match, *php_file* becomes the user (setuid). During the process of writing a new .forward file or removing an old one, it makes sure to avoid symbolic link replacement attacks³ by calling atomic functions. The user is expected to own the directory where the .forward file is located and the .forward file itself. If any of these conditions is not satisfied, the user is informed of an error, and a *syslog*(3) is sent about a possible attack.

Extensibility

By choosing PHP and its broad base of support databases and functions, much of the extensibility of

³When a cracker exploits a race condition among system services in combination with symbolic links to cause unintended effects.

this project is builtin. Using network sockets for client-server interprocess communication is trivial. Accessing a NIS encrypted password database is as simple as granting permissions to the DBM file and opening it. Likewise, LDAP and encryption functions are present. Even so, we needed to construct a loadable module to provide a few supplementary functions.

Luckily, PHP makes it particularly easy to extend itself via dynamically loadable shared object libraries. The AUcoe supplied module contains some RPC clients, a Samba hash generator, and a wrapper for Cracklib.

The Perl Samba password changing daemon is provided as a generic model for quick and dirty extensibility. It receives connections from the network using a simple *socket(2) accept(3)* loop, verifies the connection is from the Web server, reads the record from the network connection, and writes it to the private Samba *smbpasswd* file. It also does some memory caching optimizations. (see the code).

Individuals wishing to add further functions in PHP should take a look at the *ldappw.php3* file and see how all of the LDAP functionality is encapsulated into a single object. Multiple LDAP server connections can be instantiated independently. This is the model that future extensions should use, and that the rest of the code will, eventually, be re-written to use.

Limitations and Future Directions

The code has not undergone significant outside testing. It should have an *AutoConf(1)* configuration for choosing options and a better installation procedure than copying files. More of the code should be converted to a class/object interface like the LDAP framework. The password verification mechanisms could be made more generic by allowing more choices like *PAM(3)* and *nsswitch.conf(4)*.

We constructed a GUI front-end and middleware between the login and the various independent functions. This allows us to offer an easy way for users to login once at the beginning and automatically get access to the various functions (password changing, forwarding mail, access printer accounting information, generating one time passwords, etc.) However, it is currently not as comely as it might be and is fairly site specific.

We have also thought about integrating the rule-based configuration language of something like *npasswd* or *passwd+*, but given the extreme thoroughness of Cracklib, it may add marginal benefit.

Acknowledgments

This project could not have been completed without the help of various people and the support of the College of Engineering at Auburn University. Special thanks to Jerry Carter, for providing the Samba hash generation code and for some of the LDAP

integration help, and to director Stephen Henderson who always supports us in our endeavors. Thanks to the PHP core and documentation team for putting together an outstanding programming environment, the Apache team for Web server integration, and the folks at Debian for making it easy to keep all of the packages and their dependencies up to date. Also thanks to my wife for understanding my work ethic and my two year old son who provides a wonderful source of stress relief (No, no.. **you** eat it..)

Availability

The code is currently available via anonymous ftp from <ftp.eng.auburn.edu> in `pub/doug/AUcoeEw.tar.gz` or available from the author's tools page at <http://www.eng.auburn.edu/~doug/second.html>

It is currently in production beta state – it works, but needs lots of configuration via *globals.php3*. The code is known to work on Solaris2.6 and above and on Linux, specifically Debian.

Author Information

Doug Hughes received a BE in Computer Engineering from Penn State University in 1991. His first exposure to UNIX was on a Harris HCX-7 system connected to the Internet, UUCP, and the BITNET.

After graduation he worked at GE Aerospace post-RCA merger, and through the Martin Marietta merger and various smaller buy-outs. He managed to escape in 1994 (just prior to the Lockheed merger). In the mean time he gathered experience in large scale software development, systems administration, network administration, and database administration.

He worked as the Senior Network Engineer for the College of Engineering at Auburn University from 1994 until 2000, when he accepted a position with Global Crossing. At the time of publication submission he was still working for Auburn. He can be contacted electronically at doug@eng.auburn.edu (which will probably remain active indefinitely).

References

- [1] Microsoft Corporation, "Microsoft Networks SMB File Sharing Protocol (Document Version 6.0p)," Redmond, Washington, January 1, 1996.
- [2] Allison, Jeremy, "The Samba File and Print Server," *;login;*, November 1997 NT Special: 12-18.
- [3] Leighton, Luke Kenneth Casson, "Samba and Windows NT Security Interoperability," *Proceedings of the 3rd Large Installation Systems Administration of Windows NT Conference (LISA-NT)*, Seattle, WA, July 30 – August 2, 2000, Lake Forest, CA, USENIX, 2000.
- [4] Hornsby, David, Columbia Appletalk Package. <http://www.cs.mu.oz.au/appletalk/cap.html>, University of Melbourne, Australia.

- [5] Originally by Rasmus Lerdorf, *Portable Home Page*, <http://www.php4.org/>, 1994.
- [6] Muffet, Alec, *Cracklib: A ProActive Password Sanity Library*, <http://www.users.dircon.co.uk/~crypto/>, 1997.
- [7] Cottrell, Pete, "Password File Management at the University of Maryland," *Proceedings of the Large Installation Systems Administrators Conference, Philadelphia, PA, April 9-10, 1987*, Lake Forest, CA, USENIX, 1987. 32-33.
- [8] Curry, David A., Samuel D. Kimery, Kent C. De La Croix, Jeffrey R. Schwab, "ACMAINT: An Account Creation and Maintenance System for Distributed UNIX Systems," *Proceedings of the 4th Systems Administration Conference (LISA '90)*, Colorado Springs, CO, October 18-19, 1990, Lake Forest, CA, USENIX, 1990, 1-10.
- [9] Pato, Joseph N., Elizabeth Martin, Betsy Davis, "A User Account Registration System for a Large (Heterogeneous) UNIX Network," *Proceedings of the USENIX Conference*, Dallas, TX, Winter 1988, Lake Forest, CA, USENIX, 1988, 155-161.
- [10] Hoover, Clyde, *npasswd*. Last updated July 13, 1999, <http://www.utexas.edu/cc/unix/software/npasswd>.
- [11] Bishop, Matt, "Anatomy of a Proactive Password Changer," *Proceedings of the 2nd Usenix Security Symposium*, Baltimore MD, September 14-17, 1992, Lake Forest, CA, USENIX 1992, 171-184.
- [12] *ANLPassword*, Source code, Last updated Feb 13, 1995, <ftp://info.mcs.anl.gov/pub/systems/anlpasswd.tar.Z>.
- [13] Davis, Eric Allen, *Epasswd: Solving the Heterogeneous Password Program Problem*, <http://www.nas.nasa.gov/Groups/Security/epasswd/>.
- [14] Spencer, Henry, "Shuse At Two: Multi-Host Account Administration," *Proceedings of the 11th Systems Administration Conference*, (LISA '97), San Diego, CA, October 26-31, 1997, Lake Forest, CA, USENIX 1997, 65-69.
- [15] Riddle, Paul, Paul Danckaert, Matt Metaferia, "AGUS: An Automatic Multi-Platform Account Generation System," *Proceedings of the 9th Systems Administration Conference (LISA '95)*, Monterey, CA, September 17-22, 1995, Lake Forest, CA, Usenix 1995, 171-180.
- [16] Harris, J. Archer and Gregory Gingerich. "The design and implementation of a network account management system." *Proceedings of the 10th Systems Administration Conference (LISA '96)*, Chicago, IL, September 29 - October 4, 1996, Lake Forest, CA, USENIX, 1996. 181-189.
- [17] Arnold, Bob, "Accountworks: Users Create Accounts on SQL, Notes, NT, and UNIX," *Proceedings of the 12th Systems Administration Conference (LISA '98)*, Boston, MA, December 6-11, 1998, Lake Forest, CA, USENIX, 1998, 49-61.
- [18] Abbey, Jonathan, Michael Mulvaney, "Ganymede: An Extensible and Customizable Directory Management Framework," *Proceedings of the 12th Systems Administration Conference (LISA '98)*, Boston, MA, December 6-11, 1998, Lake Forest, CA, USENIX, 1998, 197-218.
- [19] Mod_Perl, <http://perl.apache.org/>.
- [20] "Automated Password Generator (APG)," Federal Information Processing Standards Publication 181, October 5, 1993, National Institute of Standards, <http://www.itl.nist.gov/fipspubs/fip181.htm>.
- [21] Apache Web Server. The Apache Software Foundation, <http://www.apache.org/>.
- [22] Spafford, Gene, Simson Garfinkel, *Practical UNIX & Internet Security, 2nd Edition*, Sebastopol, CA: O'Reilly, 1996.
- [23] Mills, David L., "Network Time Protocol (Version 3) Specification, Implementation," *RFC 1305*, March 1992.

Pelendur: Steward of the Sysadmin

*Matt Curtin – Interhack Corporation
Sandy Farrar & Tami King – The Ohio State University*

ABSTRACT

Here we describe Pelendur, a system for the management of common system operation tasks. Specifically, Pelendur focuses on the management of user accounts and related information (such as groups) across platforms and even for particular software packages (like databases) that require user authentication. Pelendur has reduced the massive process of deleting expired accounts and creating new accounts between terms from a week-long operation by several part-time operators (with subsequent cleanup by a collaboration of instructors and staff) into a completely automated process that requires less than 15 minutes of staff work and completely eliminates the need for instructor intervention.

Introduction

In 1998, our department was in the midst of a massive migration of our computing facilities wherein we moved from an architecture of many HP-UX clusters to an architecture using Solaris-based function-specific servers with thin clients in offices and labs. Some of the software used in the old environment needed to be replaced [3]. Because of severe limitations in the functionality and correctness of the largely ad-hoc scripts run by operators for the creation of accounts, it was determined that an account management system capable of managing our evolving multi-platform environment was needed.

Yet Another Account Management System?

The idea of implementing a software system for the management of user accounts is not new; past years' LISA conferences have seen many such systems. Even if we briefly ignore the issue of availability, some systems described were unsuitable because of extreme differences in the way that accounts are created and managed [1, 5] and incompatible means of handling account data [7, 11, 9, 10, 4, 8].

In the end, the most compelling reason for us to build our own software was that a grander vision existed: a single data repository for our department, which would include such things as data needed for user accounts, course-specific computing requirements, and access to various limited-access department resources. We could not find any available account management system that would work easily with any sort of database that we would construct.

The Academic Environment

Account management, though a fairly straightforward task, is quite demanding in an academic environment. Each term, we receive course rosters from the university registrar. Although students who major in computer and information science have "permanent" accounts (those that will remain until after they graduate or change majors), we have thousands of other accounts that are created specifically for the

duration of the term. We have one week between most terms, which means that during this time, we need to delete potentially more than 2,500 accounts and then create another 3,000 accounts for the next term.

In this paper, and in our environment, we use a term that will be important to understand: "section". This is a specific "class" that meets together. This term is introduced because many sections of a given class can be scheduled for the same term and we need to know the most granular level of grouping available from the registrar.

System Requirements

Specific requirements for this system were identified. In its full design, our scope is actually much more broad than the rather specific task of account management. The reason for this is that parts of account management (such as determining how much quota to associate with an account) are dependent on other criteria like the requirements for courses that the user of that account is scheduled to take. For example, a course that deals with particularly large data sets might have a requirement for more than the default amount of disk quota.

Initial requirements focused on the actual management of user accounts and dependencies.

Account Management: This is the management of individual account profiles. Adding, editing, expiring, and purging them.

Course Management Courses have particular requirements (such as the need for one platform or another and the use of a software system like Sybase) that need to be configured and managed. Some of these configuration options are simple matters of preference. Others are matters of policy, which the course coordinator does not have the authority to change. (Though we refer specifically to courses, there is nothing that prevents these entities to be managed from being project groups, departments, or any other sort of group that might have particular requirements in other environments.)

Resource Management: Anything that exists in our environment (such as “Unix machines”, “NT machines”, “Sybase database”, “disk quota”, “print quota”, and “color printers”) might also need to be managed. As these are neither accounts nor courses, but share basic properties, we classify all of these as “resources”. The basic point here is that users who are administratively responsible for these resources manage them through Pelendur instead of having us perform all of the management tasks for them.

Additional requirements were slated for future development. Some of these features are now implemented and others are still on our “to-do” list.

Unix groups: As we use groups to manage sets of users on the Unix systems, Pelendur should be aware of groups and know how to use them. (This interface is now partially implemented; new groups are created, but old groups are not garbage collected.)

Mailing lists: Some of our course instructors and students prefer to use mailing lists to stay in touch. We presently use both faculty-maintained aliases files and Majordomo [2], but are now investigating the possibility of replacing both of these mechanisms with Mailman [12].

Course directories: Because some courses have group projects or software that is specific to the course, we need to be able to associate directories with a given course. Handling of filesystem permissions should enforce the policy for read and write access established by the instructor.

Multidomain management: Currently, everything that is a part of the instructional environment is considered “the system”. Accounts that belong to individual research labs are not managed by Pelendur but it could be convenient for us to have that option available. Should we do this in the future, it would be nice to have the option of having a single Pelendur installation be able to manage multiple “systems”, rather than having to make a new installation of the software for each domain that needs to be managed. This feature would also be useful to include access to limited-access machines, such as those that are set aside for long-term computationally-intensive jobs.

Electronic lock systems: A relatively new addition to our department is an electronic lock system, whereby ID cards are used for access control instead of physical keys. Presently, this is managed by a standalone DOS-based system.

Course newsgroups: Most courses in our environment is assigned a newsgroup on our local news server. This is the default means of providing an “out of class” communication channel. Presently, we just create new groups as new courses are added and cancel the messages

in the groups at the end of each term by hand. Though this is not a big time-sink, we would like for these processes to be automated.

Course-specific environment: Some courses have particular environmental needs, such as a course-specific \$PATH setting, for example. Pelendur can provide this.

“Any computing resource”: As we continue to move forward, other resources are identified and incorporated into the system’s functionality. At a very high level, the goal of the system is to manage the systems’ configurations so that system administrators can do other things that computers can’t do very well, like planning.

Design and Implementation

Pelendur is a large system, made up of several programs. We’ll first describe the philosophies that influenced the system’s design and then consider the programs and major library modules that these programs use.

Data-Driven

The entire system sits atop a Sybase relational database. Rather than creating code that would depend upon very specific data, working with entities that make sense for our environment, we opted to put as much of the system in data as possible and to make those data be as generic and flexible as possible. Thus, rather than dealing with courses, sections, and instructors, other users of Pelendur will be able to work with teams, departments, and project coordinators. Each deployment of Pelendur will define its own terms and the relationships among them that make sense.

We believe it important to emphasize that this makes the integrity of the database especially critical. In such a highly dynamic system, we’re not dealing with simple cases of the Wrong Thing failing to achieve the desired result. Data that have been compromised by a moderately clever attacker can be used to attack the system itself, creating accounts for attackers, granting them privileges to the entire system, and possibly even running commands with superuser access.

The schema is represented in Figure 1. Here we describe each of these tables in some detail.

People contains information about a person. When a person is added to the table they are assigned a database identifier and a user login. Those two values will be used to tie a person to their resources.

Account is a table that contains information about individual accounts.

Classification contains information on thingies in the system. A thing can be anything with the exception of a person or an individual account. Most classifications define a resource or a membership group. There are also templates and defaults in Classification that are used to

create resources and membership groups or define certain values in the system (e.g., what the current quarter is). Each classification is assigned a unique number (SID) when it is added to the database.

MembershipIn contains the memberships for the membership groups from Classification. It maps a user login to a SID and is used to determine what resources that login should have.

UserRights defines owners of classifications and grants access to users for a classification. This allows a user to manage resources for their membership groups.

ResourceUsedBy defines what resources a classification has. It also indicates if the resource can be edited by the owner or proxy of the classification.

ResourceGroup links classifications together.

Flexible

We made an effort to avoid “hardwiring” anything in the system. This was largely accomplished by taking a very dynamic view of the data. That is, instead of having an “account” with a “disk quota” field that would be assigned a value based on the state of the system at the time of the account’s creation, all information about an account must go through a process of resolution, where its dependencies are determined and the values for each of the account’s properties are resolved at run-time. This view was taken for everything in the system, not just accounts.

Something else that we incorporated in order to allow maximal flexibility is a system of property inheritance. Although dealing with a relational database system at the core, we were able to provide the ability to inherit properties from a parent by specifying a relationship between various records in the database by defining a “parent ID” (Called “PID”) as a means of determining which “SID” is one step closer to the root than the current.

This gives us the ability to specify object-oriented “is-a” relationships between records in the database. Thus, the “tree” of elements to be resolved can be of an arbitrary depth, allowing each site (and each particular type of resource being managed) to have an appropriate number of levels to support the sort of abstraction desired, without forcing some high level of overhead on those who do not need to deal with such abstractions.

A good example of how we use the ability to inherit properties is in the case of a series of courses that a student will take in sequence. There will be some Classification table entry that will identify the series. Each course in the series will have its own Classification entry whose PID identifies the Classification entry of the series as the parent. Each section in a course will have its own Classification entry whose PID identifies the Classification entry of the parent course. Thus, configuration changes are made at the appropriate level: those that affect all sections in the

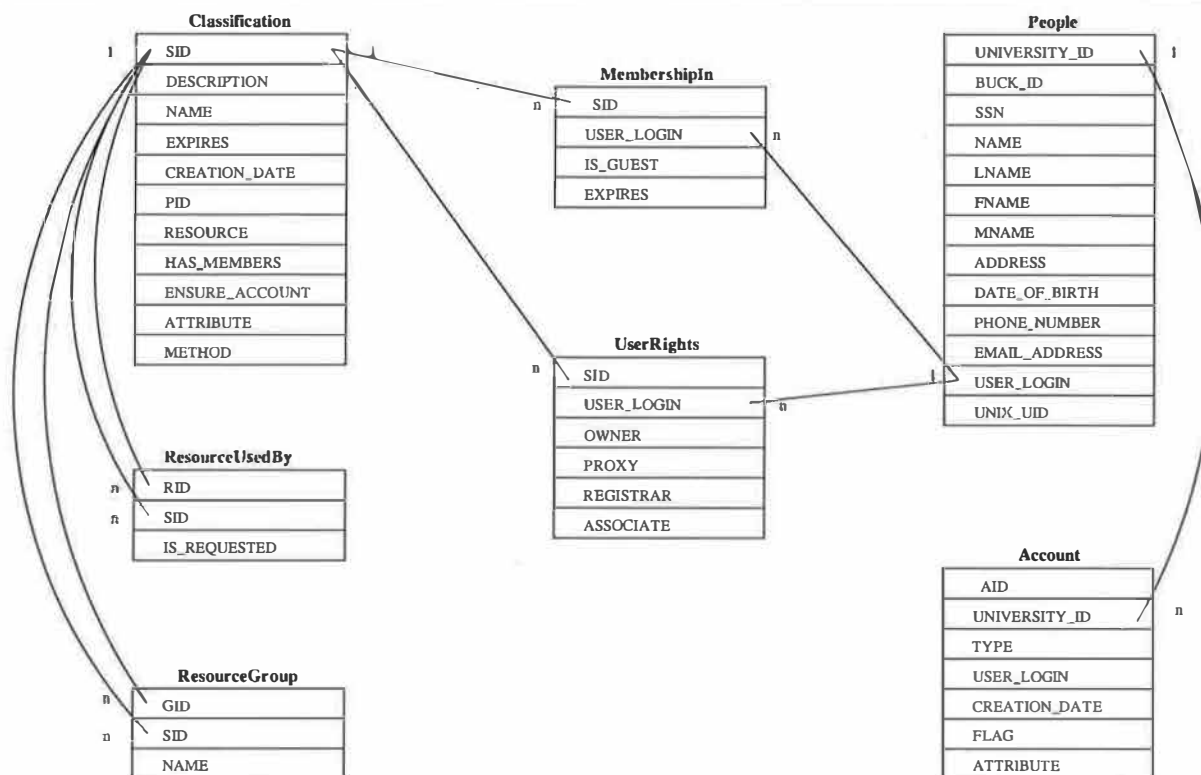


Figure 1: Schema of Pelendur's Database

series will be made in the Classification entry for the series, those that affect all sections in a specific course in the series will have those changes made in the course's Classification entry, and those that are specific to a particular section will be made in that section's Classification entry. When we need to determine how much quota, for example, an account has, we'll determine which sections in which the account has membership. Those will be resolved by walking up the tree until we get to the root object (as determined by having a PID of 0), populating the fields in memory with the values in the database and returning. By the time the initial Classification entry returns, we will have queried each level in the tree, populating the object with the levels specified at the highest level first, and overriding those with whatever (if anything) was specified in the lower levels.

Additionally, where values are numeric, they need not be absolute; signed numeric values indicate relative values. Consequently, a property of a Classification might be to increase disk quota by 30MB, instead of specifying some absolute value.

Modular

Rather than requiring code changes in many different places in order to create new interfaces in the system, we designed the system to be made up of a basic core which includes the database and the resource resolution logic. The rest of the system interfaces to that core. The core understands when an account needs to be created, or what the properties of an account at any given time should be, but it knows nothing about creating accounts. Instead, it can tell what the account's properties are and what system(s) need to reflect those properties. If the account needs access to both Unix and Sybase, the core will pass the appropriate information to the modules for Unix and Sybase through a standardized interface.

As described in section on implementation language, this design makes it possible for even the database itself to be replaced with another database that has the same schema.

Implementation Language

Perl was chosen as the implementation language. Its freely available DBI (database interface) and DBD (database driver) package for talking to a wide variety of databases makes it an excellent option for building atop a database:

- Very little investment is made in an interface to any particular database (since we code to DBI), thus allowing us (theoretically) to use any database for which a DBD module exists. In practice, we did use one Sybase-specific feature, which we can and will in the future stop using.
- We can spend our time focusing on the problem at hand, instead of how to write to the database.
- Being free, the price is right.

- Since source code is available, if there is a problem with it for which a fix is not available, we can fix the bug ourselves.

Because parts of this system will need to run with superuser privileges, we're concerned about safety of our code. Perl has some excellent safety features: namely, the ability to identify "tainted" data and support for arrays and buffers that grow dynamically.

Finally, Perl is available on a huge number of platforms. As long as we keep portability in mind, Perl will provide us all of the language support necessary to allow our code to run unmodified on essentially any system we could use in the foreseeable future.

Programs

rosterload is the program that loads rosters into the database. In our environment, rosters are course rosters that identify which students and instructors should be associated with a given section. For the most part, rosterload just calls **Roster::rosterload**, which does all of the work. Our plans are for rosterload and Roster.pm to be modified to be able to load roster information directly out of the Data Warehouse or from email.

makeaccounts is the program responsible for creating things, that is, resources and accounts. First it will initialize the resource methods, then it creates any resources that don't already exist. Then it creates any accounts that don't exist. We have a cron job run makeaccounts runs twice per day.

expireaccounts is our garbage collector; it's responsible for removing things from the system. Expired resources are removed from both the database and from the systems that were used to support the account. After this has been accomplished, it will remove entries in the "MembershipIn" table that have expired and then any user resources that are specified for the user in the database. cron runs expireaccounts once per day.

notifier sends account removal notifications. All accounts scheduled to be removed will receive a seven day notification. Account that are 'persistent' will also receive a 30 and 14 day notification of expiration.¹ Note that this includes not only accounts for operating systems, but this also includes "accounts" in software systems like Sybase. As Pelendur continues to help us blur the distinctions among different systems that comprise the "computing environment", we'll move away from providing notification that specific systems' accounts will be unavailable and provide notification only on the user's "meta-account" in the department. The Sybase

¹The amounts of time on the notification are configurable parameters.

account will be created as soon as the student shows up on the roster for a class that has this resource. The account will always² match the Unix login name and will have the typical default password.

crconfig is the course configuration interface for instructors. This provides a convenient means for them to manipulate the database in a controlled manner, allowing them to change only that which is under their administrative authority without creating artificial restrictions that require staff to perform their administrative tasks for them. Instructors and their “proxies” (those whom they designate) to add and to remove resources from a particular section and a course default. Project and course directories are also configured through this interface.

cradmin is the account management administration tool. Essentially, this is the interface that is used to manipulate the state of the database on anything that is in the database. Where **crconfig** deals with abstractions and has a “course-specific” view of the world, **cradmin** allows manipulation of non-course-related resources. The user interface itself still works with many of these abstractions, but it is in this program where we can make changes to the user interface to allow administration of new resources.

Modules

Here we describe the major modules of the Pelendur system. Many of these modules are shared by various standalone programs in the system.

IICFDB.pm is the module contains the generic routines for interacting with the account management database. There is typically a routine for each table for searching, adding, and removing. They follow the naming convention `get_<table-name>`, `add_<table-name>`, `remove_<table-name>`. Table names are converted to lowercase and underscores are used where there would be white space (“MembershipIn” becomes `membership_in`). For some tables an update function exists also. The `get_` routines are all polymorphic and will do different searches depending on what data is passed to the routine. This module also contains the routines to walk the database recursively (`resolve_pids`) in order to resolve all dependencies and provide an up-to-the-second view of an account’s properties, as determined by what the database knows about the account.

IICFLog.pm is our logging mechanism. Presently, this basically accepts messages and puts them

²“Always” is a pretty strong word. There are a few exceptions, as the result of ancient accounts that predate Pelendur that still contain dashes (-) so they can’t be used as logins to Sybase. If the username has a dash in it, the dash will be converted to an underscore (_) for the Sybase account. New accounts are always created with names that are portable across our systems so that these kinds of conversions will not be necessary.

in the “right place”, but the intention is that it will be a general-purpose log gatherer for all applications in our environment.

IICFLogin.pm contains the routines that deals with logins in our environment. It contains the routines that generates new usernames and default passwords.

NT.pm contains the routines to do all things on the NT systems. It is currently not implemented; an older standalone account creation and deletion system was developed locally for NT accounts. Instead of implementing this part of the system initially, we opted to focus on the Unix, Sybase, and core database portions of the system, building an interface between Pelendur and the old NT account management scripts. This decision has turned out to work relatively well for us, and has saved us from what could have become a large amount of redundant work as Microsoft seems to think that making major interface changes from version to version of its operating systems is an appropriate thing to do. Thus, we can avoid writing most of NT.pm until the NT based systems are on a newer version of the operating system than the current programs support.

PQuota.pm interfaces with the printing system’s quota handling. We use LPRng [6] for our printing system throughout the department.

ResourceMethods.pm is a module that contains all of the methods for the resources in the account management system. Each resource has its method defined for it in the `METHOD` field in its “Classification” entry. When called, this method will ‘do the right thing’ for the resource. For example, a resource like “mailing list” might have a method that specifies a program to be run in order to create or to delete the mailing list.

Roster.pm contains the routines for processing the roster.

Sybase.pm contains all of the routines for dealing with Sybase resources.

Unix.pm contains all of the routines for Unix resources. It adds and removes accounts in the password file, manipulates the group file, and generates the Quotas file for disk quotas.

mkcisdir.pm is used to create and to remove directories on the Unix systems. It handles several types of directories: users’ home directories, group project directories, and the directories used by Submit, our program for electronic laboratory submissions. Where possible, this module will run as the owner of the directory instead of root.

The Effect of Pelendur

Our environment has benefited tremendously from Pelendur in many ways. What used to be a

painful experience for all is now essentially a non-event.

Labor

The total staff labor expenditure for processing accounts between terms is greatly reduced.

- Accounts to be removed are no longer processed manually. When an account has no more references (managed through the Membership table), the account is garbage collected.
- Instructors would manually add and remove students from their sections using hardcopy provided by the university registrar. We now get these data from the registrar directly and automatically add and remove students.

Error Rate

Historically, this has been a problem. The old software had to run by someone who knew its idiosyncrasies and limitations. Mistakes were frequent and could easily require several hours to fix. Errors are now much less frequent, because we get data directly from the registrar and do not require any manual intervention before account configuration. Because the entire state of the system is driven by the database, errors can now be fixed by making appropriate changes in the database and waiting for Pelendur to propagate them.

Since managing our systems with Pelendur, we have been able to identify accounts that have expired long ago but were never removed, to identify problems that arise because of an account being misclassified, and generally to free ourselves of the kinds of concerns that come about when the administrators need to manage things manually.

Latency

In section "Programs", we identified which parts of the system run on a regular basis in our environment. These are configurable to a site's requirements. In our environment specifically, this means that changes made take no more than one day to take effect. This is a huge difference from the days, weeks, or even more that it took under the old system.

Future Work

Quite a lot can still be done with Pelendur. Specifically, we need to increase the number of systems against which we can interface, including native NT account management, more intelligent

Conclusions

Account management can be greatly simplified by taking a more abstract view and thinking of system access as a property that results from the state of the account. Pelendur has proven to be a highly effective means of managing a very large number of highly variable accounts.

Availability

Although the system has been designed and implemented in a way that emphasizes flexibility and

freedom from very a very site-specific view of the world, it will still take quite a bit of effort for another site to bring Pelendur into production. We're currently working on finishing the functionality and hope that we will be able to revisit some of the areas of the system that work for us but would make it difficult or impossible for other sites to use the system as-is. This work is geared toward making a general release of the system. We have no idea when this could possibly take place.

Author Information

Matt Curtin is founder of Interhack Corporation, which helps developers and system managers build and run systems they can trust. He is also a Lecturer at The Ohio State University's Department of Computer and Information Science. His current interests include Lisp programming, secure systems development, and Internet privacy. Reach him electronically at cmcurtin@cis.ohio-state.edu.

Tami King is a senior software specialist for the Computer and Information Science Department at The Ohio State University. Since graduating from Utah State University with a B.S. in computer science, she has worked as programmer, UNIX systems administrator, and database administrator. She also managed a group of UNIX systems administrators before returning to programming, what she enjoys most. Her free time is taken up by her husband, her new baby boy, and pedaling out of sync. Reach her electronically at tami@cis.ohio-state.edu.

References

- [1] Arnold, Bob, "Accountworks: Users create accounts on SQL, notes, NT, and UNIX," *Twelfth Systems Administration Conference (LISA '98)*, page 49, USENIX, Boston, Massachusetts, December 6-11 1998.
- [2] Chapman, D. Brent, "Majordomo: How I manage 17 mailing lists without answering "request" mail," *Systems Administration (LISA VI) Conference*, pages 135-143, Long Beach, CA, USENIX, October 19-23 1992.
- [3] Curtin, Matt, *Creating an Environment for Reusable Software Research: A Case Study in Reusability*, Technical Report OSU-CISRC-8/99-TR21, The Ohio State University, Department of Computer and Information Science, August 1999.
- [4] Geer, Daniel E. Jr., "Service Management at Project Athena," *Large Installation Systems Administration Workshop Proceedings*, page 71, Monterey, CA, USENIX, November 17-18 1988.
- [5] Harris, J. Archer and Gregory Gingerich, "The Design and Implementation of a Network Account Management System," *10th Systems Administration Conference (LISA '96)*, pages 33-41, Chicago, IL, USENIX, September 29-October 4 1996.

- [6] Powell, Patrick and Justin Mason, "Lprng – An Enhanced Printer Spooler System" *Ninth Systems Administration Conference (LISA '95)*, pages 13-24, Monterey, CA, USENIX, September 17-22 1995.
- [7] Riddle, Paul, Paul Danckaert, and Matt Metaferia, "AGUS: An Automatic Multi-platform Account Generation System," In *Ninth Systems Administration Conference (LISA '95)*, pages 171-180, Monterey, CA, USENIX, September 17-22 1995.
- [8] Rosenstein, Mark A., Daniel E. Geer, Jr., and Peter J. Levine, "The Athena Service Management System," *USENIX Conference Proceedings*, pages 203-211, Dallas, TX, USENIX, Winter 1988.
- [9] Spencer, Henry, "Shuse: Multi-host Account Administration," *10th Systems Administration Conference (LISA '96)*, pages 25-32, Chicago, IL, USENIX, September 29-October 4 1996.
- [10] Spencer, Spencer, "Shuse at two: Multi-host Account Administration," *Eleventh Systems Administration Conference (LISA '97)*, page 65, San Diego, USENIX, California, October 26-31 1997.
- [11] Tomas, Gregory S., James O. Schroeder, Merrill E. Orcutt, Desiree C. Johnson, Jeffrey T. Simmelink, and John P. Moore, "UNIXhost Administration in a Heterogeneous Distributed Computing Environment," *10th Systems Administration Conference (LISA '96)*, pages 43-50, Chicago, IL, USENIX, September 29-October 4 1996.
- [12] John Viega, Barry Warsaw, and Ken Manheimer, "Mailman: The GNUmailing List Manager," *Twelfth Systems Administration Conference (LISA '98)*, page 309, Boston, Massachusetts, USENIX, December 6-11 1998.

Network Information Management and Distribution in a Heterogeneous and Decentralized Enterprise Environment

Alexander Kent & James Clifford – Los Alamos National Laboratory

ABSTRACT

To promote enterprise-wide information and resource sharing, we have implemented a network information management and distribution system that gives subscribing systems real-time access to relevant information changes. Participating systems need little more than a small, single application to receive the updates. User interaction and data administration require only a web browser. The resulting system is timely, reliable, secure, easy to support and maintain, and extensible.

Introduction

In Los Alamos National Laboratory's distributed and heterogeneous computing environment of more than 12,000 users and more than 20,000 computers, the task of maintaining accurate network services information is a complex undertaking. Information such as network accounts, e-mail addresses, login names, aliases, home-page pointers, privileges, permissions, and so on, is constantly changing as people come and go or change job assignments. Matters are further complicated when the input data comes from different sources and must then be distributed across multiple server systems. To simplify the system administrator's job, we designed a system to make it easier for network services to get the current and accurate information they need to do their job. We call it A Real Time Information Management and Update System (ARTIMUS).

If you have attended past USENIX LISA conferences or read the proceedings, the problem and solution may sound vaguely familiar. The general requirements for an account management system have been nearly the same for over ten years, including: [2]

- Vendor and service independence
- Data flexibility
- Minimal changes to existing software
- Automated account installation
- User access to data

This paper focuses on the unique aspects of the Los Alamos National Laboratory (LANL) approach. Specifically, these areas include the database design, manipulation of data within the central database, propagation of that data to end-hosts and services, and security. Recent systems differ on whether they use a "big hammer" relational database or not. ARTIMUS, [1], [4], and [11] do; [2], [7], [9], and [10] do not.

Motivation

Our organization, LANL's Network Engineering Group, is the Lab's Internet service provider. We

provide the usual services: LAN and dialup connectivity to the Internet, DNS, white pages directory information, authentication, e-mail accounts, web page publishing, and so on. Other LANL organizations offer compute, storage, print, personnel, training, library, and many more network services to internal and remote customers. In addition, organizations throughout the Laboratory run their own network servers to satisfy local needs. Each organization, and even individuals, choose the type and configuration of computing system(s) that best meets their needs. While this looks like a recipe for chaos, it also presents opportunities.

The group needed a sane way to manage people, accounts, and other network information for our ISP business. With the right design, we saw that we had a chance to help the other network service providers too. Subscribers could participate in the same centralized account and information management system that we use on their own servers. These are the requirements we felt we had to meet to have a viable system:

- Each user should be able to make changes to his or her own network information. A third party, like a system administrator, should not be needed to make the change.
- The end user interface should be intuitive and consistent. Managing your network profile will not be a frequent or familiar activity. The goal here is to allow changes to be entered quickly and accurately without a call to the help desk.
- Existing subscriber system software should not require modification. Programs, either shrink wrap or open software, will still run using existing data formats and access methods.
- Additional subscriber systems should be easy to integrate. We will not own or manage many of these clients. Changes to subscriber systems must be few and minor in nature.
- Subscriber systems should not be affected by changes to our data structures and business rules.

- Subscriber systems should not be affected by failures in the information management system.
- Subscriber systems should be notified of data changes in near real-time so users can view and test changes immediately after making them.
- Subscriber systems should get only accurate, consistent, and current data.
- System administrators should retain control over their systems. For example, they should be able to control who gets access and privileges on their systems. There should also be provisions for system administrators to make local changes outside of ARTIMUS.

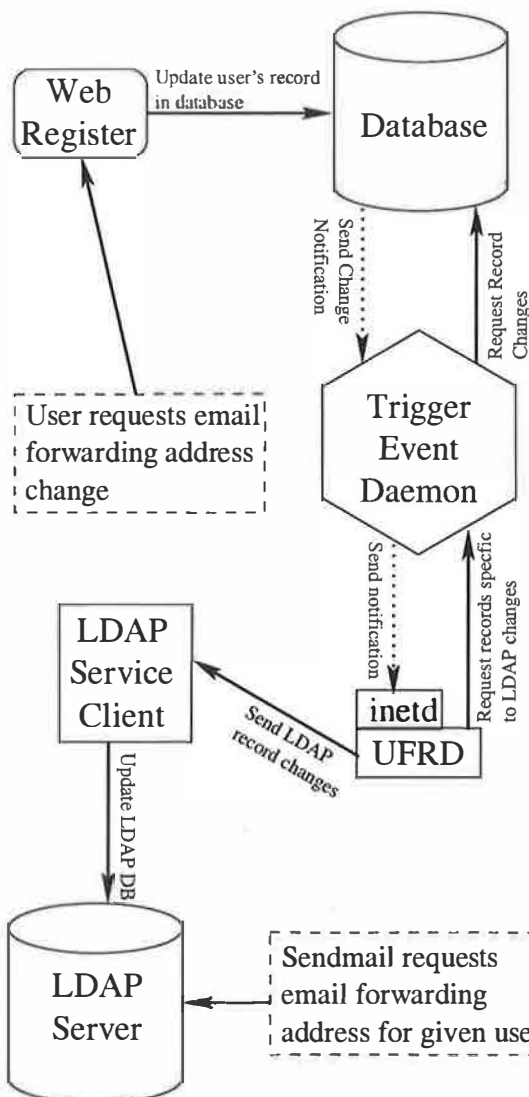


Figure 1: Example data flow for a user setting a new e-mail forwarding address for a given name, and sendmail requesting the forwarding address.

- System administrators should see their workload drop after participating in ARTIMUS.
- ARTIMUS should be extensible. The difficulty in adding new services and operating systems should be in step with their complexity.

- ARTIMUS should have high availability.
- ARTIMUS should import corporate data to eliminate redundant data entry and promote consistency. For example, employee data from HR could be used to build organizational e-mail lists.
- ARTIMUS should be secure. Information should be readable and modifiable only by authorized persons.
- ARTIMUS, not subscriber systems, should implement site policies and business rules.

Overview and Example

Here is a simplified example to demonstrate ARTIMUS' data flow, which can be seen in Figure 1. Suppose Amy wants mail that is sent to the institutional address `help@lanl.gov` to be delivered to her departmental server account `amy@cic-mail.lanl.gov`. Here is how it is done: Amy uses her web browser to go to the Network Registry service and logs in to identify herself. She creates `help` as a new name which the Network Registry inserts into the database assigning ownership to Amy. Then she sets the forwarding address for `help` and the web server completes the task by updating the database. The change to the table containing names triggers a database procedure that makes a copy of the updated record and notifies the Trigger Event Daemon (TED) process (see the section "End Service Propagation"). TED notifies the subscriber services affected by the forwarding address change so those systems can update their local copies of e-mail forwarding information.

System Design

ARTIMUS is a three-tier design of web-based user interface to central database to end-system propagation. The system is functionally divided into these three distinct segments to provide maximum security, flexibility and simplicity.

Web Interface

Users view and change information with ARTIMUS' web interface called the Network Registry. All user modifiable information is updated in this manner, traditional applications are not used (e.g., `chfn`). This GUI interface is designed to be friendlier than native system interfaces. In addition, the SSL web interface encrypts the data between the desktop and the web server thus protecting the registration information from capture and modification.

User access to ARTIMUS data is controlled through the web interface. The user must first authenticate to the Network Registry with an employee identifier and password. The web interface then enforces access controls that restrict what the user can change within the database. Normally users may only change information they "own." However, system administrators may be granted access to change other users' information. We considered pushing the authentication

and access control into the database itself but Sybase, which we currently use, does not support any appropriate external authentication systems like RADIUS or Kerberos.

In the previous example of Amy changing the forwarding address of `help@lanl.gov`, Amy goes to the URL for the Network Registry and authenticates with her employee identifier and password. Selecting the name `help` she then enters the forwarding address as seen in Figure 2. The Network Registry then does a SMTP VRFY on the given forwarding address and submits an update to the database name table if the address is valid.

In addition to e-mail forwarding, the Network Registry is used to create and remove user accounts on UNIX and Microsoft-based systems, change passwords, and manipulate several other user network attributes. Additional features and systems are constantly being added to the Network Registry.

The web interface is built upon whiz [8], a Python-based sequential CGI forms tool that provides a simple, stateful, and authenticated method of managing the registry segments and adding additional features. Communication to the database is handled through a locally developed system called `sqld` which wraps database traffic with SSL-based encryption between the Network Registry and the database. Other, more standard database access libraries could be substituted for `sqld`.

The Database

The authoritative copy of nearly all network service information is kept in a relational database. The Network Registry and various corporate repositories, like human resources' employee data, are ARTIMUS' information sources. The database enforces syntax, consistency, various business rules, and serves as a central information clearinghouse.

If the information is accurate in the database, it will be accurate on the servers. And if it is wrong in the database, it will be wrong everywhere. Because data integrity is so important, we use the database's features such as syntax checking on table fields, inter-table consistency enforcement, and "triggers" that call stored procedures to validate record changes. Both Sybase and Oracle products support these features. Previously, we tried maintaining consistency by doing the checking in the web server and other applications that modified the database. It worked but changes to table structures and field definitions were more difficult because several programmers had to modify code simultaneously. This explains our move to a "big hammer" relational database.

The database tables are divided into three primary categories: one set for *person* data, one for *name* data, and one for *authentication/authorization* data. *Person* data is needed for ownership; i.e., some *person* owns each table entry in the database. *Person* data is also needed for white-pages publishing. A *name* has an owner and may have attributes such as a forwarding address, a URL, a UNIX UID, sharing group members, and mail list members. *Authentication* and *authorization* data include a *person's* passwords, authentication tokens, accounts, and permissions or authorizations.

A portion of the network information database is shown in Figure 3. Designing the database tables was a non-trivial, iterative process. Subscriber requirements for data, syntax, and business rules were gathered and database design principles were applied [6]. The resulting design was reviewed by peers and customers. Then the process was repeated as more services were added and mistakes were uncovered.

Here is how the syntax, integrity, and business rules are applied in Amy's e-mail forwarding address example. Before `help` is added to the name table, the

User Registration

Add Account

Set Forwarding
Address

Change Mail
Server Password

E-Mail List
Manager

Delete Account

E-Mail Options -- Set Forwarding Address

Choose the name you wish to forward e-mail for:

Enter the forwarding address:

Figure 2: Network Registry page for adding/changing e-mail forwarding addresses.

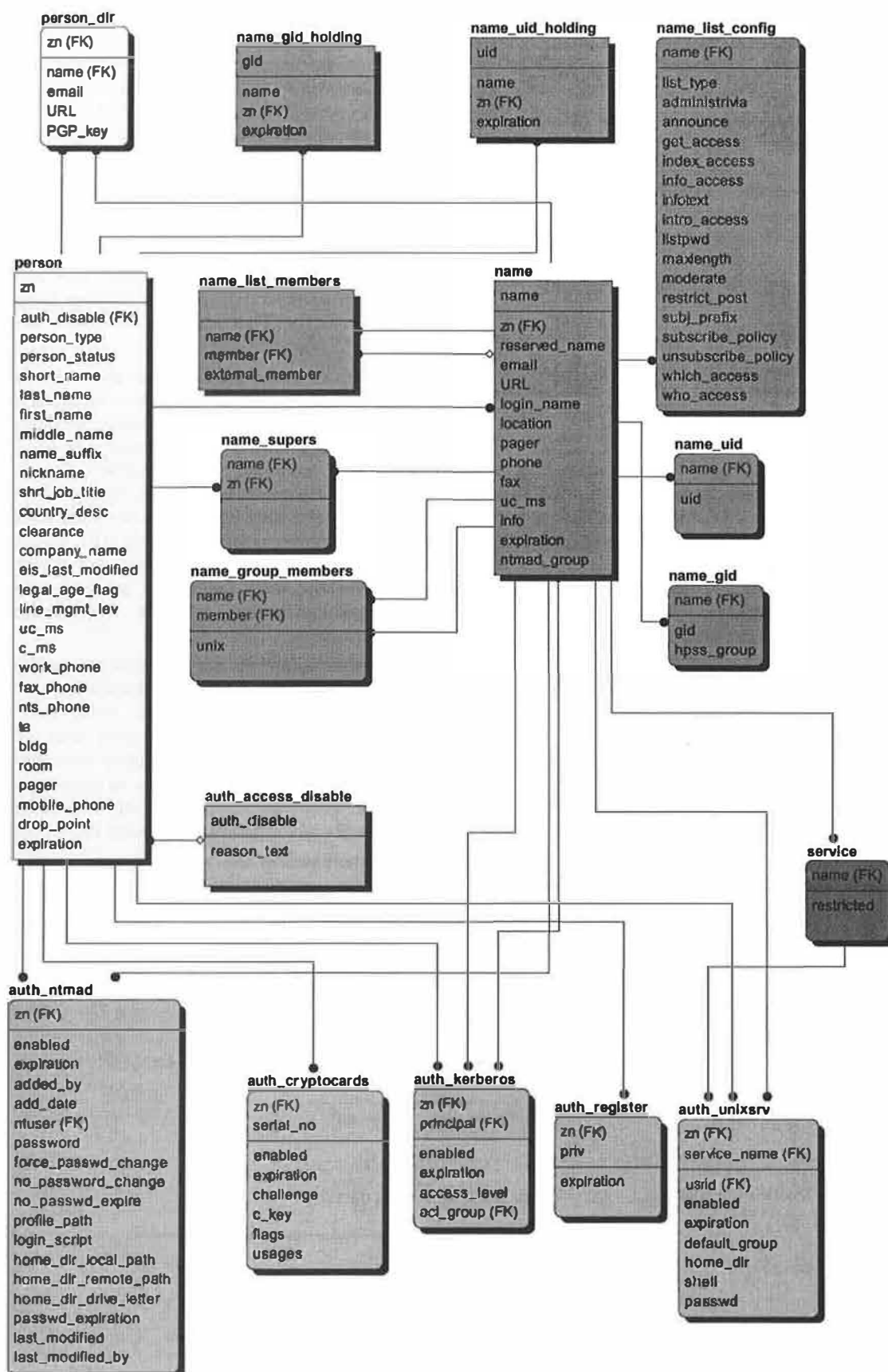


Figure 3: Major tables and dependencies within the database.

length and characters are checked. Names with @'s or spaces are rejected.¹ Also, names that already exist are rejected to enforce uniqueness.

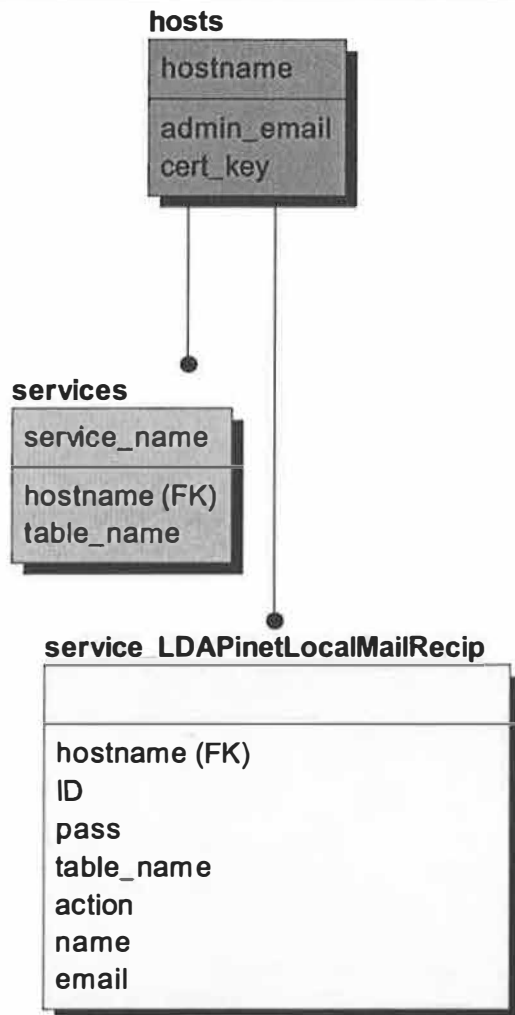


Figure 4: TED support tables within the database.

Similarly, the e-mail forwarding address, amy@cic-mail.lanl.gov, is checked before it is added to the database. This time an @ is required.

The effects of applying the concepts of business rules and data integrity are more striking in the example of an employee leaving. Our first rule is to wait for seven days after someone's personnel record disappears before deleting any data. Data entry mistakes happen and resurrections are simple.² When a personnel record disappears, that individual's expiration date is set to today plus seven days in the person table. Later, a cron job initiates deleting expired people. Before a person is deleted, the database deletes all of his or her names. The SQL command delete from name

¹Names cannot have a @ because an implicit @lanl.gov is appended when associated with a forwarding address.

²While not actually removing any information, all authentication and authorization (UNIX, NT, etc.) accounts for the person are disabled immediately.

where zn='Amy'³ deletes all of Amy's names. But before a name can be deleted, the database also cleans up anything that depends on that name. For example, when help is deleted, e-mail lists are cleaned up with delete from name_list_members where name='help'. This cascaded cleaning keeps the database internally consistent and, even better, reduces bounced e-mail to list owners.

End Service Propagation

Propagating information from the database to the subscriber systems is a critical component of ARTIMUS. The in-house written subsystem that performs this function is affectionately called *TED* and *UFRD*.

The Trigger Event Daemon (TED) feeds database changes to the Update Fields Real-time Daemons (UFRD) running on subscriber systems. UFRD passes the changes to local programs that then process the updates. See Figure 1.

The TED-UFRD subsystem is somewhat similar to the Project Athena Zephyr Notification System [3] in terms of providing immediate, reliable, and high fan-out information propagation. However, the TED-UFRD design is simpler and specific to transmitting information from a central source, the database, to end systems, the subscribers. Data transmission security and subscriber system data access control are built in.

The TED-UFRD system disseminates information changes in near real-time from database tables to a list of subscriber hosts affected by the changes. Instead of requiring services to wait for cron jobs to run or other polling mechanisms, the end services are immediately notified of information updates. Oracle has a similar notification facility but requires Oracle software on each subscriber system [5].

Currently LANL updates the following services through TED-UFRD with an average of 1500 information events per day:

- RADIUS dial-in passwords for the institutional modem pool (1150+ users)
- CRYPTOCARD one-time password token card accounts (7200+ users)
- Microsoft master accounts domain for the institution (4400+ users and global groups)
- LDAP white-pages system (19,000+ record objects)

Additional services will begin using TED and UFRD over the next few months, including UNIX account and group management, Kerberos, Entrust, and DCE/DFS systems.

Trigger Event Daemon

Moving data from the database system to TED starts with triggers on tables containing information needed by the subscriber services. In the e-mail forwarding address example, the trigger is on the name table. A trigger fires (i.e., calls a stored procedure in

³zn is a unique person identifier, usually an employee number. Amy is used here for simplicity.

the database) when a table is changed. The database allows one trigger per action (add, update, and delete) per table so TED triggers must coexist with a table's integrity triggers.

To remember table changes, TED triggers create new records in TED service tables with the type of modification (add, delete, delete-update, or add-update), the name of the table being modified, a

```

create trigger name_trg
  on name for insert
as
-- Begin TED update
  declare @tbl_name          varchar(40)
         ,@name              typ_name
         ,@zn                typ_zn
         ,@reserved_name     typ_flag
         ,@email              typ_email
         ,@URL                varchar(255)
         ,@login_name        typ_flag
         ,@location          varchar(16)
         ,@pager              typ_pager
         ,@phone              typ_phone
         ,@fax                typ_fax
         ,@uc_ms              varchar(4)
         ,@info               varchar(255)
         ,@expiration         datetime
         ,@ntmad_group        typ_flag
  select @tbl_name = "lanl..name"
  declare ted_crshr cursor
    for select * from inserted
  open ted_crshr
  ted_loop:
    fetch ted_crshr into @name,@zn,@reserved_name,@email,@URL,
      @login_name,@location,@pager,@phone,@fax,@uc_ms,
      @info,@expiration,@ntmad_group
    if @@sqlstatus = 0 begin
      exec TED..send_service_LDAPmail_fwd(@tbl_name,
        "ADD",@name,@email)
      goto ted_loop
    end
  close ted_crshr
-- End TED update

```

Figure 5: Insert trigger attached to name table for updating e-mail addresses.

```

create proc send_service_LDAPmail_fwd
  @table          varchar(40)
  ,@action         typ_action
  ,@name           typ_name
  ,@fwd_addr       typ_email
as
  declare host_crshr cursor
    for select hostname from TED..services
      where service_name=@service_name
  open host_crshr
  host_loop:
    fetch host_crshr into @host
    if @@sqlstatus = 0 begin
      insert TED..service_unixsrv values(
        @host,0,@table,@action,@name,@fwd_addr)
      goto host_loop
    end
  close host_crshr
  exec TED..notify("LDAPinetLocalMailRecip")
  return 0

```

Figure 6: Stored procedure called by triggers attached to name table to move records to TED tables and notify TED daemon.

subscriber host name, and the modified fields. A separate record is inserted into the service table for each subscriber host needing the information. The service table for e-mail forwarding and two other TED support tables are shown in Figure 4. The trigger sends a UDP packet to TED with the name of the service table where the inserts were placed. To simplify the coding, TED's triggers actually call a stored procedure to insert the records into TED service tables. The trigger for e-mail forwarding attached to the name table can

be seen in Figure 5. The stored procedure called by the trigger is in Figure 6.

Upon receiving the UDP packet, TED reads the indicated service table from the database.⁴ Then TED sends a UDP packet to each subscriber host it finds in the service table records. The UDP packet signals the host(s) to start UFRD. The host's UFRD makes a TCP

⁴TED will work with multiple database servers. As a security measure, it will only connect to database system(s) in its configuration file.

```
#!/usr/bin/perl
use Net::LDAPapi;
use Sys::Syslog;

# Update inetMailRecipient object.
# Changes read from stdin look like:
# table|action|name|fwd_address
$local_domain = "lanl.gov"
$sep=chr(255); # Separate fields with char 255
$ROOTDN = "cn=root, o=Los Alamos National Laboratory, c=US";
$ROOTPW = "oob";
$ldap_server = "ldap.lanl.gov";

&openlog("ufrd.LDAPmailRecipient",'pid','daemon');
&Sys::Syslog::setlogsock('unix');

$lid = new Net::LDAPapi($ldap_server);
if ($lid == -1){
    &syslog("err","Connection to $ldap_server failed");
    exit 1;
}

if ($lid->bind_s($ROOTDN,$ROOTPW) != LDAP_SUCCESS){
    &syslog("err","LDAP bind error: $lid->errstring");
    exit 1;
}

while (<>){
    chomp;
    ($table,$action,$name,$fwd_addr)=split($sep);
    $ENTRYDN="cn=$name, objectClass=inetLocalMailRecipient";
    # Treat update add/deletes same as regular add/deletes
    if ($action =~ "DELETE" || $action =~ "UPDEL"){
        if ($lid->delete_s($ENTRYDN) != LDAP_SUCCESS){
            &syslog("err","LDAP delete error: $lid->errstring");
            exit 1;
        }
    }
    elsif ($action =~ "ADD" || $action =~ "UPADD"){
        %ldapdata = ("cn" => $name,
                    "objectClass" => "inetLocalMailRecipient",
                    "mailLocalAddress" => "$name==[@]==$local_domain",
                    "mailRoutingAddress" => $fwd_addr);
        if ($lid->add_s($ENTRYDN,%ldapdata) != LDAP_SUCCESS){
            &syslog("err","LDAP add error: $lid->errstring");
            exit 1;
        }
    }
}

$lid->unbind;
exit 0;
```

Figure 7: UFRD client that creates/modifies/deletes LDAP inetLocalMailRecipient objects.

connection to TED and reads its information updates. When a change is successfully processed, UFRD sends TED an acknowledgement and the matching service table record for that host is removed. A TED service table record is kept until an acknowledgement is received so failures with UFRD can be retried.

To handle subscriber hosts that do not request their information updates, TED periodically reads all service tables' unprocessed data records. For each system with pending updates, TED sends another UDP notification since the subscriber host may have been down or unreachable. To summarize, information change notifications are sent to subscriber systems in near real-time and resent until successfully processed and acknowledged.

Because the information is stored in the database, state is maintained even if TED or the UFRD clients should terminate for unexpected reasons. TED will send an e-mail notification to the administrator listed in the TED's hosts table if a subscriber has unprocessed requests over two hours old.

Update Fields Real-time Daemon

A UDP packet from TED starts a UFRD on a subscriber host; on UNIX systems UFRD is started by `inetd`. The packet contains a port number to connect back to on the TED system. UFRD first reads its configuration file to find the encryption key to use with the TED system that sent the UDP packet. UFRD then makes a TCP connection to the TED server and sets up DES3 encryption using the key it found. UFRD receives the pending information updates and, based on the table name field, executes the appropriate command to process the data. The information update line(s) are passed to the command as standard input under UNIX and message passing under Windows NT. If the command exits successfully, UFRD indicates to TED to remove the corresponding record(s) from the TED service table.

The commands called by UFRD to update information may be written in the most convenient method: C, C++, Perl, Python, shell script, or some specialized interface for the service. The Perl code to process e-mail forwarding address changes is in Figure 7.

UFRD has been ported to several operating systems including Linux, Solaris, UNICOS, Irix, and Windows NT. Its simple functionality relying on running native commands to update local data allows quick porting to new platforms.

Future Plans

The group has several future improvements planned for the ARTIMUS system.

Extending ARTIMUS to other network information systems is foremost. Generating dynamic DNS updates is a goal for the next year. Our host information is already maintained through a web interface and kept in a Sybase database but does not use ARTIMUS.

The difficult part will be reconciling the names' owners.

Offering the service to other Laboratory organizations has begun and will be expanded. Given the flexible and secure ARTIMUS framework, it is easy to provide central account management services to network servers throughout the Laboratory. Adding a Network Registry module, a few database tables, constraints, and triggers, as well as writing an update command for UFRD service to call, is not difficult.

The Network Registry continues to grow. Currently, we are redesigning much of the look and feel in attempt to make it easier and more intuitive for the end-users; user interface design is non-trivial. Various business and integrity checks continue to be added, both within the Network Registry and the database.

ARTIMUS needs additional rules to prevent inappropriate bulk changes from happening. Occasionally, we receive a large, but bogus, data feed from the corporate data warehouse that must be detected and rejected.

We would like to get real-time updates propagated from the Lab's corporate sources to pass along to end customers. Such real-time linkage would allow new employees to request accounts upon arrival (and have them created in real-time as well, thanks to ARTIMUS), plus automatically disable authentication and authorization accounts the moment they are terminated in the human resource system.

We would like to examine the use of individual user accounts within the database and direct authentication to the database. The tradeoffs of security benefits versus complexity may or may not make this practical. For various security issues we are considering moving from Sybase to Oracle and using Oracle's Kerberos and RADIUS authentication. Oracle's DBMS ALERT package may also be superior to the current UDP-based database notification mechanism for TED.

Additional issues, ideas, and plans will come up as we add new services and think up new methods to use the system.

Conclusions

The ARTIMUS design, including its web interface, relational database with built in data integrity, real-time update service, and data protection provides a solid base for eventually managing nearly all of the Laboratory's account and network information data. We are optimistic the implementation will meet or exceed the requirements listed in the Motivation section and discussed below. Ultimately, we will measure the system's success by the number of subscribers it supports.

User Interface Requirements

Users can manage their own network information through the Network Registry. Web based

applications with authentication are commonplace at Los Alamos. Most customers will be able to easily maintain their own network profiles. Keeping the navigation simple, intuitive and consistent as features are added will be a challenge.

Subscriber System Requirements

Adding subscriber systems to ARTIMUS requires new entries in the TED tables, a UFRD daemon, a UFRD service application, a configuration file, and a shared DES key. The total installation time is usually under an hour.

Currently, 99 percent of the changes to the database are propagated to the subscriber systems in under two seconds via TED-UFRD. We expect the mean response time and standard deviation to both shrink when we retire some frequently run ad hoc database query applications.

System Administrator Requirements

There are a few system administrator features in ARTIMUS that are outside the main theme of this paper. We will briefly mention them here. Common system names like "root" are reserved, as are UNIX UIDs and GIDs under 1000. OS, third party, and local software that define users and groups can coexist with ARTIMUS. System administrators can define their subscriber systems as open or closed. Users can add accounts to open systems but only system administrators or their designees may add accounts to closed systems. Finally, we have defined organization administrators who can manage most information for anyone in an organization, and name administrators who, in addition to the owner, can change any of a name's attributes.

ARTIMUS Requirements

ARTIMUS already supports UFRD on a variety of operating systems (Figure 8) and services (Figure 9) but not all services are supported on all OSs. The first UFRD for a UNIX system took about a month to write and debug. The first UFRD for Windows NT also took about a month. Bringing up subsequent UNIX UFRDs is proportional to the time it takes to locate the necessary system include files and libraries.

UFRD agent scripts as seen in the LDAP e-mail address example (Figure 7) are usually simple and straightforward to write. The approach is: read standard input; split the line into an array (or list); and execute some existing application to add, delete, or modify some local data file. A Perl program to manage Linux user accounts by calling adduser, deluser, and moduser is 36 lines long.

Microsoft Windows NT	Microsoft Windows 2000*
Linux	Sun Solaris
SGI IRIX	Compaq Tru64 UNIX*
IBM AIX	Cray/SGI UNICOS

* under development

Figure 8: Operating systems where UFRD runs.

The current ARTIMUS implementation is reliable. We do health checks on our services every few minutes throughout the day. When a service is down for five minutes or more, an on-call person is paged. The last six months of web, database, and TED error logs show only four transient failures. Connecting to the web server failed twice and connecting to the database also failed twice. There were no prolonged failures resulting in an alarm page.

Since subscriber systems depend on ARTIMUS for updates, downtime is only noticeable to users wanting to make changes. Clearly, the main thing to avoid is losing the database. Backups and mirroring are both done.

Oddly, overall network service availability can actually be improved by adding a central database. LDAP and RADIUS are important services to the Laboratory. ARTIMUS makes replicating these servers as easy as deploying a system and adding a hostname to the TED host and service tables.

Data integrity is a key feature of ARTIMUS. It is also the most complex to implement. But, when done correctly, it promises the biggest rewards from the system. Our most difficult and time consuming problems are usually caused by incorrect, incomplete, or inconsistent data on one or more server systems. ARTIMUS

E-mail forwarding addresses (aliases)	E-mail lists*
URL's	Directories for web and FTP publishing*
UNIX accounts	Sharing group membership*
UNIX UID and GID's	Windows accounts and sharing groups
LDAP white pages	LDAP yellow pages*
LDAP POSIX users and groups*	LDAP mail recipients
CRYPTOCARD token cards	RADIUS accounts and passwords
Kerberos accounts*	DCE accounts*
Authorizations*	DNS resource records*
SecurID token cards*	Entrust PKI certificates*

* under development

Figure 9: Applications that currently have UFRD service interfaces.

prevents these problems by simply rejecting bad information before it goes into the database.

Finally, security is an essential part of the ARTIMUS design. This system would not be deployed at Los Alamos if it did not protect users' network information.

Availability

Due to U. S. encryption export regulations, release of the source for the TED-UFRD propagation system must be controlled. Those interested in obtaining the source code may contact the authors directly to determine the necessary arrangements. Licensing restrictions beyond the export issue are otherwise liberal.

The web pages and database currently include many LANL idiosyncrasies. They would probably require significant changes to work elsewhere and therefore are not available at this time.

Acknowledgements

The authors would like to thank Los Alamos' Network Services Team. ARTIMUS is a product of the entire team's hard work and expertise. Particular recognition goes to Mary Gentry for her extensive work on the database design and implementation and her extraordinary commitment to detail. Susan Buchroeder was responsible for the Network Registry design. Amy Meilander provided the excellent database figures and lent us the use of her name(s) in the examples throughout the paper. Thanks to Jerome Heckenkamp, Elise Lee, and Giridhar Raichur for their constructive criticisms and thoughtful editorial comments. Finally, we are grateful to our supportive manager, Kyran Kemper.

Author Information

Alexander (Alex) Kent is a Systems Software Engineer for the Network Engineering Group at Los Alamos National Laboratory. His primary development projects include Laboratory-wide authentication and user account systems, network information propagation, and the Los Alamos firewall system. In addition, he is a full time graduate student at the University of New Mexico nearing completion of an MBA. Alex has a BS and MS in CS from New Mexico Tech. He may be reached via e-mail: alex@lanl.gov or snail mail: MS B255, Los Alamos, NM 87545.

James (Jim) Clifford is the Network Services Team Leader and a Systems Software Engineer for the Network Engineering Group at Los Alamos National Laboratory. His interests include Internet technology, Linux, and practical computer security. Jim has a BS from the University of Michigan. He may be reached via e-mail: jrc@lanl.gov or snail mail: MS B255, Los Alamos, NM 87545.

References

- [1] Bob Arnold, "Accountworks: Users Create Accounts on SQL, Notes, NT, and UNIX," *12th Systems Administration Conference (LISA'98)*, pages 50-61. USENIX, 1998.
- [2] David Curry, Samuel D. Kimery, Kent C. De La Croix, and Jeffery R. Schwab, "ACMAINT: An Account Creation and Maintenance System for Distributed UNIX Systems," *Workshop on Large Installation System Administration*. USENIX, 1990.
- [3] C. A. DellaFera, M. W. Eichin, R. S. French, D. C. Jedlinsky, J. T. Kohl, and W. E. Sommerfeld, "The Zephyr Notification System," *Usenix Conference Proceedings*, USENIX, 1988.
- [4] Jon Finke, "Invited Talk: Manage People, Not Userids," *10th Systems Administration Conference (LISA'96)*, USENIX, 1996.
- [5] Jon Finke, "Oracle tricks and techniques in supporting systems administration," *System Administrator and Network Security Institute (SANS)*, 2000.
- [6] Candace C. Fleming and Barbara von Halle, *Handbook of Relational Database Design*, Addison Wesley, 1989.
- [7] J. Archer Harris and Gregory Gingerich, "The Design and Implementation of a Network Account Management System," *Usenix Conference Proceedings*, USENIX, 1996.
- [8] Neale Pickett, "Whiz," <http://starship.python.net/crew/neale/src/whiz/>.
- [9] Paul Riddle, Paul Danckaert, and Matt Metaferia, "AGUS: An Automatic Multi-Platform Account Generation System," *9th Systems Administration Conference (LISA'95)*, pages 171-180, USENIX, 1995.
- [10] Henry Spencer, "Shuse: Multi-Host Account Administration," *Usenix Conference Proceedings*, USENIX, 1996.
- [11] Gregory S. Thomas, James O. Schroeder, Merrill E. Orcutt, Desiree C. Johnson, Jeffrey T. Simmelink, and John P. Moore, "UNIX Host Administration in a Heterogeneous Distributed Computing Environment," *Usenix Conference Proceedings*, USENIX, 1996.

xps: Dynamic Tree Watching under X

Rocky Bernstein – Breakaway Solutions

ABSTRACT

The xps program dynamically displays the Unix processes as a tree or forest in an X Window, the roots on the left and the leaf processes (those with no children) on the right. The status of each process running, sleeping, stopped, etc., can be indicated by different colors. Different users can appear as different colors too.

Process selection can be made per user, all users, or through a regular-expression pattern.

In contrast to the terminal-based `ps`tree or tree-widget based programs, the tree display uses diagonal lines, and effort is made to effectively use the full 2-dimensional area of the screen by balancing levels and centering the children of a node between their parent. A goal of the program is to give an idea of what's going on graphically as things may be constantly changing. Therefore the display algorithm tries to keep processes close to their parents to reduce the amount of scrolling to see localized process creation and destruction. Some effort is also given to make sure that the tree layout doesn't get wildly reorganized when there are small or localized changes. This makes it easier for the eye to pick up and recognize the changes over a potentially large display area.

We describe here criteria for tree animations such as this one and how the xps layout algorithm works.

There are some other miscellaneous features of xps. One can select viewing the processes by a single user, a regular expression for users, by all users, and perhaps show kernel processes. One can click on a process to get more information (via `ps` or a user-specified program) about that process, send a signal, or set the process priority, assuming you have the permission to do so.

Since programs of this ilk can consume a bit of CPU on their own, some effort has been made to turn off the update process when the program is iconified or not visible for some other reason such as being obscured by another window. Some attention has been paid to make algorithm display fairly fast in most situations, although it has to be admitted that this comes sometimes at the expense of a nicer layout.

What Is xps? Why xps?

There are a number of front ends or GUIs that perform various underlying commands, and sometimes one wants to see what's going on behind the scenes. Some examples include:

- Learning more about what is going on in configuration and installation processes; `make` or `GNU configure`, for example
- Tracking down zombie creation
- Understanding the processes for a user or contained in a process group; these generally cluster into subtrees
- Helping distinguish a process via its lineage. For example one may have many `bash` sessions running. Some may be spawned from `sshd`, some from `telnet`, some via `emacs` or an `xinit` session.

To glean what's going on, many systems administrators use `ps` or `top` or a `top` variant, like `gtop`. The `ps` program can list the process id and its parent process id. However it is sometimes useful to display the parent-child relation graphically.

Considerations for Doing a Tree Layout for xps: Tree Animations

xps uses a custom tree-layout algorithm for positions nodes and draws lines between them using low-level Xlib calls. Probably each time a new graphics library came out I considered ditching the custom tree-layout routine with a generic tree package. However to date, I've not found an acceptable one.

Tree widgets are often used for browsing file systems or menu systems. Generally these objects are fairly static. There is generally a way to collapse or expand a branch in the tree; this makes sense when information doesn't change all that much. But in the context of xps, sometimes the new or changed information is precisely what you want to see. So instead, in xps, symbolic or filtering rules are used to focus the display, rather than by zooming in or out from a user-selected tree branch. In particular, one can select the area of interest by a regular expression which is matched against the process-owner name.

In trees with menus or files, the display is pretty much static, and therefore the time users spent to

customize the display may be cost effective; it may not be as important to spend time initially making connections between tree objects immediately visually distinct. Convention or experience seems to indicate that it is acceptable to use a Manhattan-metric line connection style, that is, where tree lines either go horizontally or vertically. See Figure 1 for a tree layout using Manhattan-metric lines.

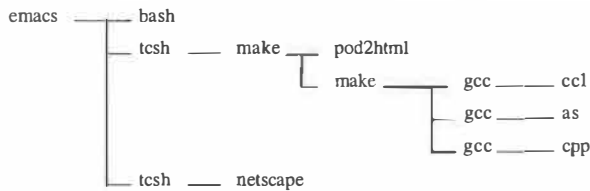


Figure 1: Lopsidedness and space-wasting layout using a Manhattan-metric tree widget.

The tree-layout requirements of xps are perhaps a bit more difficult to satisfy. In fact xps might be better thought of as an animation rather than a static charting program, and I am not aware of much literature on doing animations with trees or forests.

For an analogy, consider the differences between viewing a still-life painting and an animated cartoon. The cells of an animated cartoon do not have to be as perfectly rendered as a painting. Instead it is more important in an animation to make the *series* of cells relate to convey a story or action.

Below we give a simple example of the kinds of problems faced.

In a world where things don't change, like a figure in a book, most people will find it most pleasing for graphs with one or two children to look as in the left-hand part of Figure 2 rather than the left-hand part of Figure 3.



Figure 2: Aesthetic tree layout when things don't change.



Figure 3: Possibly better tree layouts when trees shift between one another.

However, now consider the case where things are constantly changing. Suppose we want to chart what is going on when we have a process that forks a process, *child1*, and then forks a number of other processes sequentially. For a moment there will be *child2*;

it dies, and then suppose a moment later a new process, *child3*, is spawned. Suppose that now runs for a short period of time and dies; so we are back to the single *child1*; then a new process *child4* starts up and so on.

In this scenario, if you render the one or two child processes as in Figure 2, you will see a bit of flicker. This is distracting and does not assist comprehending what's really going on: that a new process is being started and finished while nothing is happening to *child1*. If instead the graphs were rendered as in Figure 3, the display is more pleasing to the eye *in animation*; it is less spastic and more comprehensible even though the left-hand or right-hand side in isolation may not be the most esthetic rendering of the graph.

In sum, hysteresis of layout can help visualization.

Now consider the differences between a Manhattan-metric layout of the variety one often sees with a tree widget, and one that uses diagonal lines in an artificial but not uncommon situation as is often seen in xps.

Notice in Figure 1, how the root of the tree, *emacs*, is very far removed from its bottom-most node. This is not a problem with the Manhattan-metric connection style per se. However almost all most tree widget programs do this kind of layout; to set the position of a node, the layout algorithm can be extremely simple since it doesn't have to consider the positions of the children.

In addition to the lopsidedness, there's a bit of space that is wasted between *make* and *netscape* that doesn't appear in Figure 4. Furthermore, the blank area to the right of *bash* and *pod2html* is reclaimed reducing the overall dimensions of the graph. However, the more-compact layout is at not at the expense of readability.

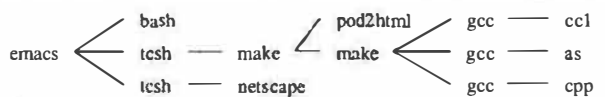


Figure 4: Layout. Nodes are more centered around their children; diagonal lines improve readability. Layout makes better use of available space and therefore dimensions are reduced.

Finally, compare Figure 1 and Figure 4 to see how the use of diagonal lines helps readability.

Experience has suggested these display criteria for a tree-animation program such as xps:

- The layout needs to be fast – it is performed every second
- The layout should be compact
- The layout should have hysteresis – reduce flickering which is annoying to watch and makes finding important changes hard to spot
- The layout should be “pretty” on the display so users can understand the relationships quickly

We now go into some of these needs in more detail.

Fast Layout

In designing any monitoring program, such as xps, a cardinal rule should be: don't trash the system you are trying to monitor.

Because layout needs to be fast and work for a large number of nodes, positioning decisions should be pretty much linear. Currently one pass from root to the leaves is made. I've considered making a backward pass as well – leaves to roots.

The program has a number of hacks to avoid layout recalculation when it is not needed. It checks to see if processes have changed; if not, nothing is changed. However this is tricky since xps shows the status of each process (e.g., whether it is running or sleeping or stopped), and if this alone changes some redisplay of the node color (not position) is needed. If the display becomes iconified or fully obscured, no layout is computed.

Pretty Layout

Experience has shown that it is preferable to draw straight lines for parent – child connections rather than diagonal lines since diagonal lines take longer to draw and can appear jagged unless antialiasing is used. (Antialiasing also generally takes more time to compute and draw.) On the other hand, when needed, I find diagonal lines are easier to follow than Manhattan-metric lines.

Similarly, I've found that lining things up horizontally and vertically helps.

Layout Heuristics

The key to a program like this is its tree-forest display algorithm. In the previous section we described desirable qualities; in this section we'll give a rough idea of the actual heuristics used. In the next section we go over how this is implemented and the time complexity of the layout algorithm.

Trees tend to be narrow at the root and get bushy as one moves towards the leaves. Therefore the approach used in xps is put nodes into fixed levels (which run horizontally) as we move from the roots to the leaves (left to right here).

The algorithm we use pretty much makes one pass to keep things fast. We keep track of the maximum number of nodes at a given level as we move from roots to leaves. As the breadth increases, the gap between levels decreases up to a minimum threshold. When things get too tight, the overall dimensions of the graph increases; in display, the tree breadth is shown in the vertical direction.

Figure 5 gives an example of how levels are redivided as we move deeper in a tree.

Putting things in levels increases the likelihood that parent-child nodes will line up and that they will

stay lined up over time. Actually, although it might be useful to have xps take into account old positions of node as discussed in a previous section, right now it doesn't; it is just an artifact of the way the layout occurs that this tends to happen.

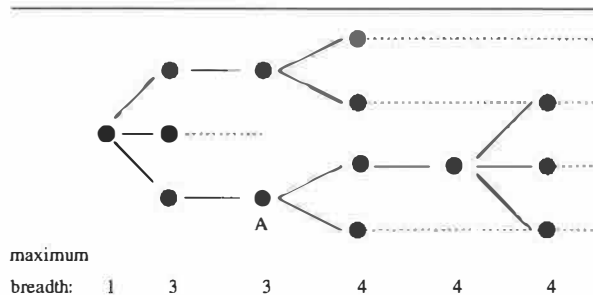


Figure 5: Tree layout along maximum-breath levels. Dotted lines show the current levels that are in effect for layout at subsequent levels. The node labeled A has rank 2 and virtual rank 3.

After creating the levels for a given depth, the dotted lines in Figure 5, xps next positions the node for that depth. To this end, the program keeps track of its level position or *virtual rank* in that depth, and compares it to the virtual rank of its parent. The virtual rank differs from the number of nodes placed so far (or *rank*), when we've decided to leave a level slot empty so as to position a child closer to its parent. In Figure 5, the node marked "A" has a virtual rank of 3, while its rank is 2. This is because we've moved the node down to its level slot 3 so it will line up with its parent.

Note that since the number of levels only increases, and when this happens inter-level space between level generally decreases, if the virtual rank of child is less than or equal to the virtual rank of its parent, it can be positioned no further down than its parent.

When setting the position of a node, xps tries to position the virtual rank of the middle child close to or less than the virtual rank of the parent, but not so much that the overall breadth is increased.

Figure 6 shows what goes on here. In the upper-left diagram we show the virtual ranks and before readjustment; to the right of that, after the readjustment at the new depth. In the bottom-left we see what happens when the new level is attached to a node further down; since centering the children would increase the overall breadth and dimensions of the tree we do not center around the middle child. The bottom-right graph show a heuristic not employed by xps but might be if a backward pass were made: readjusting the parent.

There is one layout problem that should be mentioned and is shown in Figure 7.

We see here that in drawing lines from the end of a node, nodes with a short name can sometimes cross over a neighboring node name. Currently xps does not

try to prevent this from happening. We have experimented with doing a layout as in Figure 8 without much success. Most of the time, though, the lines do not go through process labels.

Layout Algorithm implementation

The program needs to do a topological sort. A depth-first search is done to assign depth number.

The roots are then sorted by uid and pid. There is generally a small number of roots, often one. At depths deeper than the root, nodes are first arranged by attachment to a parent.

Within the children of a node, sorting is done by a bubble sort. Many people with computer science

training cringe when they hear this, because they have been taught a bubble sort takes $O(n^2)$ time in the worst case, while there are many sorting algorithms that take $O(n \log n)$ time in the worst or average case. However, a bubble sort has an advantage our most sorting algorithms: when used on *almost* sorted data, the running time is linear.

In the context of xps, the processes are often pretty much sorted by process id. The bubble-sort code we use does a check to stop early when the items are fully sorted. Also, it should be noted that the number of items to be sorted is often relatively small, since the sort is performed only on children of a node. A bubble sort generally has less overhead than most

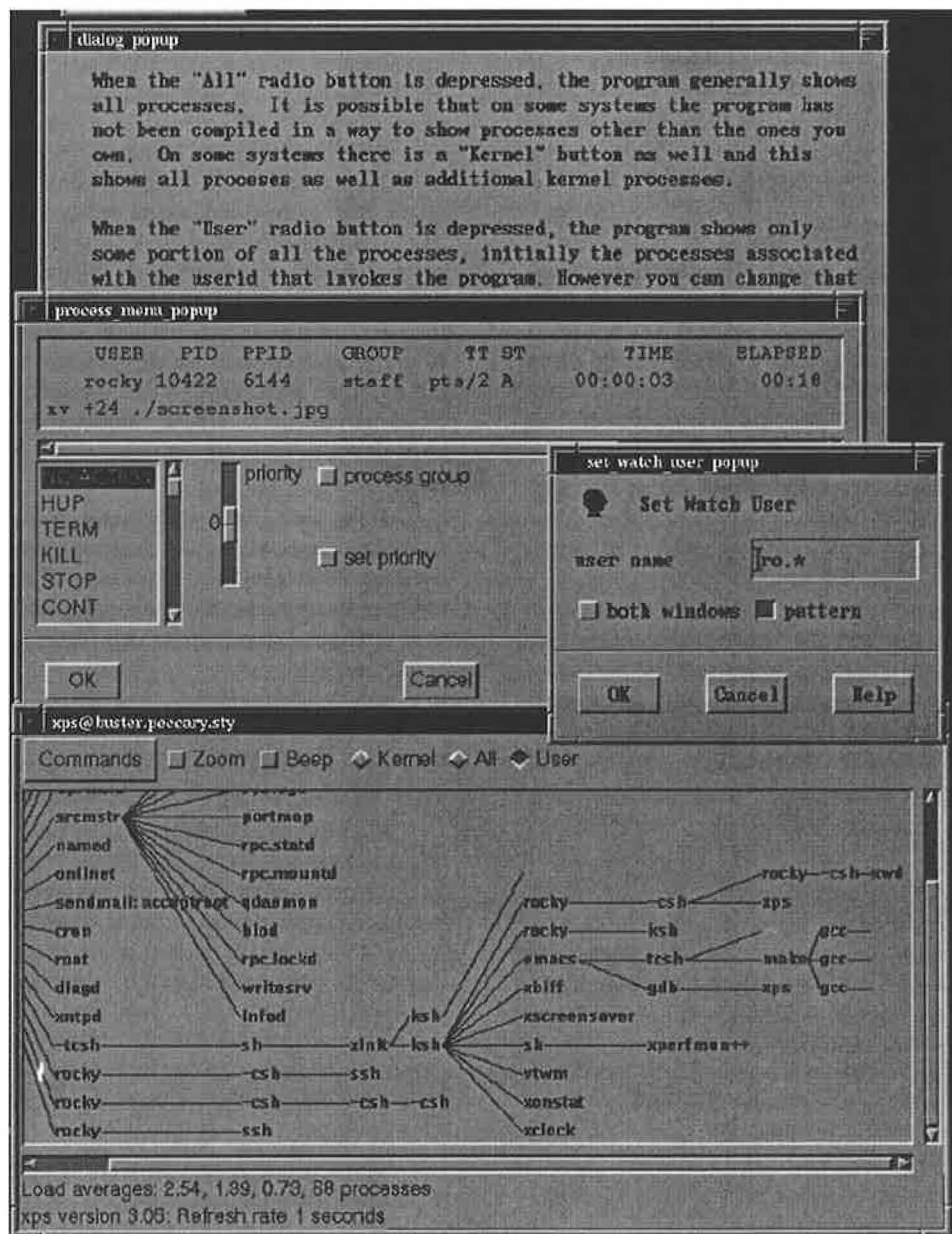


Figure 9: Aesthetic tree layout when things don't change.

other sorting programs. Still, a check on the number of children should be made and a sort like quicksort could be used if the number of children exceeds some threshold like 40 (which is in my experience rare).

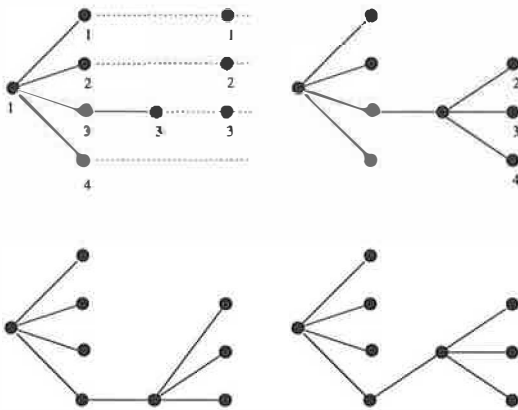


Figure 6: Example showing positioning heuristic.

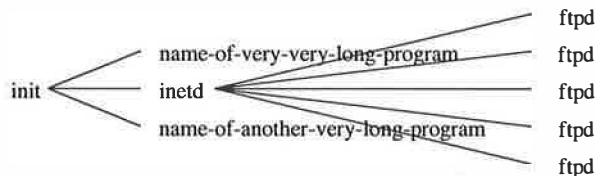


Figure 7: Crossover problems. This can be avoided by starting the vertex after `inetd` further to the right. Also note that following these lines would be easier if the difference in slopes is increased.

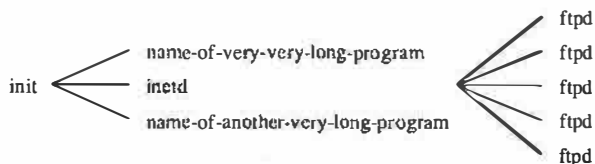


Figure 8: Crossover problems addressed by moving the drawing point further out. Increased sloped of lines may also increase readability. Compare with Figure 7.

The sorting to arrange things into levels by parent is $O(np^2)$ for n nodes, p non-leaf nodes, and where c is the maximum number of children a parent has. This is a rough analysis and looks unpromising, but in practice is probably pretty good. I suppose a radix sort could be done on the levels. Then we'd have $O(n + pc^2)$.

Still someone might want to time various algorithms. The program is not linear. Faster tree-layout and arranging algorithms would be helpful in handling larger trees.

Other Niceties of xps

xps does a sort by user id and by process id within that. This tends to group related processes together and the process id arranges things by age. It

has been suggested that if it is desirable to sort precisely by age, that should be done instead of sorting by process id.

xps has the ability to filter out processes by user id or userid – regular expression. Each user is assigned a different color. Reducing the amount of display has a two-fold benefit: it not only unclutters the display but it also reduces the amount of time needed for display.

Finally, xps has the ability to point and kill. (The interface hasn't been developed to the point of arcade games, but still it might give the systems administrator a heightened feeling of power; in contrast to mass kill programs like `kill`, it can be satisfying to see the process die before your very eyes.)

A screenshot from the program is given in Figure 9.

Future?

Just about everything could be improved. The distribution contains an extensive list of things to do. Above were some suggestions for how tree layout might be improved.

The thing I would most like to see is xps ported to the KDE qt and the GNOME gtk+ and glib libraries. Any volunteers?

Display is done by drawing on a canvas. This is primitive. Process names are not widget labels, just X strings drawn on a canvas. Figuring out the process under the mouse and showing which process is selected is a bit low-level and not tool-like. Currently, a horizontal and a vertical linear search is done to find the process id under the mouse. Binary search might be faster.

Alternatively, if a full-fledged widget for the nodes of a tree were used, then X (or an X-toolkit) would worry about when the widget is selected; presumably it uses an algorithm at least as good as binary search. Making the selected node look, well, *selected* would then be done by the toolkit.

It would be nice to benchmark the various toolkit approaches for efficiency.

One might experiment to compare speed and intuitiveness of using a canned tree widget versus a hacked layout customized for this application. Personally, I prefer the tree layout, but the code is not toolkit idiomatic.

Currently, the program contains coordinates of nodes before its position gets updated. However nothing else related to the history of the layout is saved, such as how long a node has been in the same position. It might be interesting to experiment with algorithms which make incremental improvements over time using say local heuristics. Analogous is the splay-tree algorithm which tends to make a tree balanced over time by making small localized changes as nodes are accessed.

Acknowledgements

Derik Lieber wrote the original version of this program.

Many people have read and made helpful suggestions on this paper including Stuart Frankel, Ph.D., George MacDonald, and Mike Welles.

Related work

pstree by Werner Almesberger found on Linux distributions shows a static Manhattan-metric process tree relation using character-oriented graphics.

Another program named pstree, by Lars Christensen, does about the same thing and runs on other versions of Unix. Find it at <ftp://ftp.thp.Uni-Duisburg.DE/pub/source/>.

I. Herman, G. Melançon, M. S. Marshall “Graph Visualisation and Navigation in Information Visualisation” can be consulted for a survey on graph visualization and navigation techniques, used in information visualization. For example it cites the classic paper on tree layout, E. M. Reingold and J. S. Tilford, “Tidier Drawing of Trees,” IEEE Transactions on Software Engineering, SE-7(2), pp. 223-228, (1981).

However a number of subsequent papers including those found in the survey given above question the tree heuristics used in this paper. For xps, I have not found this paper all that useful, although it is an interesting read. See the survey at <http://www.cwi.nl/InfoVisu/Survey/StarGraphVisuInInfoVis.html>.

Finally George MacDonald’s treeps program is similar. Find a home page at <http://www.slip.net/gmd/tps/treeps.htm>.

Availability

See the project’s home page at <http://www.netwinder.org/rocky/xps-home>. The program is available from <ftp://netwinder.org/users/r/rocky/xps.tar.gz>.

Author Information

Rocky “Falling squirrel” Bernstein left the University of Maryland with two bachelor’s degrees and then attended the Stevens Institute of Technology where he earned a master’s degree. Rocky has worked in a number of institutions, such as the City University of New York, IBM Research, NASA Goddard Institute for Space Studies, The Associated Press (AP), and – currently – at “Breakaway Solutions.” His e-mail address is rocky@panix.com.

Extending UNIX System Logging with SHARP

Matt Bing & Carl Erickson – Grand Valley State University

ABSTRACT

System messages in a UNIX system are handled by syslog. The responsibilities of syslog are to filter and disperse program generated messages based on a priority code contained in each message. Filtering with priority codes is not sufficient to generate enough usable information for the system administrator. Utilities which do regular expression parsing of syslog messages typically do not run continuously and thus are limited by a lack of state in detecting potentially important patterns in syslog messages.

SHARP (Syslog Heuristic Analysis and Response Program) improves the monitoring of systems by extending the existing syslog infrastructure with programmable modules. These modules use a library with a simple API to perform near real time analysis based on the messages they register to receive. System administrators can use SHARP to improve the services provided by their systems without the need for constant manual evaluation of message logs. The SHARP system and several modules were tested in a higher education production environment during the spring of 2000. Experience with SHARP indicates that it is stable, reliable, and improves the overall operation of a laboratory while not significantly increasing the workload on the system administrator.

Syslog

The “system logger”, or *syslog*, gives programs a standard interface to report interesting events to the administrator. These messages are read by a background daemon and routed accordingly. The data which a program passes to syslog is called a *message*. A message consists of two parts: priority and textual data [9]. The *priority* of a message also contains two parts: an encoded *facility* and *level*. The facility of a message is a general category into which the message fits. The level of a message is a way for the program to rate the severity of the message, typically ranging from *emerg* to *debug*. The textual data of a syslog message is a string provided by the program that describes the event being logged.

Programs use library calls to send a syslog message. The library allows the program to choose a facility and level pair, as well as the text describing the message. The library will typically prepend information such as a timestamp, hostname, program name, and PID [7]. The library then delivers the message to the syslog daemon.

The syslog daemon, *syslogd*, acts as the router for system messages. When it receives a message from a program, it in turn must decide what to do with the it. Most commonly this action involves writing the message to disk, but other potential actions include printing it to the system console, notifying online users, or forwarding the message to another system. *syslogd* makes these decisions based on a configuration file written by the system administrator. The rules in this configuration file are based entirely on the priority of the message.

Shortcomings of syslog

The standard syslog daemon¹ lacks many important features. These features impact the the reliability of message delivery and the integrity of messages after delivery.

There is no standard structure for writing syslog messages. A cleverly written program could bypass the syslog library calls and write directly to the listening *syslogd* socket. When *syslogd* reads this message, it will prepend default priority information and route the message according to these defaults [8]. While the lack of message structure is not critical for system operation, it does not encourage good programming form.

When syslog messages are forwarded over a network the problem of time synchronization arises. There are techniques for network time synchronization, but even short skews of seconds are a problem. When *syslogd* receives a message from a remote host, it writes the message to disk as it was received and does not provide an additional timestamp. When analyzing messages, this lack of consistent timestamping makes strict ordering impossible.

Some versions of syslog have the ability to route messages based upon regular expression filtering. This allows greater discrimination and handling of messages than is possible with priority filtering. Sophisticated classification and processing of messages is still difficult with regular expression filtering. Extending syslog in this manner violates the UNIX design

¹This refers specifically to the 4.4 BSD implementation [8].

philosophy of simple tools doing one thing well. Extra processing must be performed with each message, which decreases message handling capacity.

When the syslog system is compiled, the different facilities and levels are hard coded into the system. Most systems have eight different local facilities to give the programmer a set of alternative facilities. Some programs assume that a single local facility is available for exclusive use when this is not necessarily the case. With as few as eight local facilities available, there may be conflicts between programs. There is no way for the programmer to extend this limit without significantly altering the architecture.

When syslogd writes a message to a file, all priority information is lost. After the message has been written, syslogd drops the message, forever losing potentially important forensic data. The priority of the message could be useful in that it shows the state of the program that generated the message. While the text of a message is the most immediately helpful portion, it should not be treated as the only valuable piece.

After the message has been written to a file, any person with write access to the file, authorized or not, may alter the contents of the file. While there is no way to stop any intruder with total access to a system, there are cryptographic protocols designed to detect alterations in log files [3]. These protocols write a

sister log file containing cryptographic information to verify the integrity of the messages.

There is currently an IETF working group [2] that is attempting to create a new standard to solve many of these problems.

Extensions to syslog

Shortcomings in syslog and the need for improved message analysis have spawned several utility programs. These programs read syslog messages, perform analysis, and direct the results. These tools are used by administrators to automate analysis of messages.

The simplest and most common type of message analysis is the grep-style filter. This type of program regularly parses syslog messages written to disk for predefined messages. In its simplest form, this program is a shell script spawned from cron that uses regular expressions to grab messages from a log file [6]. These types of utilities have value, but only as post mortem tools. By the time the administrator has seen the output, a significant amount of time may have occurred where a potential problem may no longer be fixable. Imagine an administrator runs a regular expression filter nightly, only to find out the next morning that the company database has been filled for hours, rendering the system unusable.

Another type of tool does the same style of parsing, but in real-time. These utilities, swatch [1] being

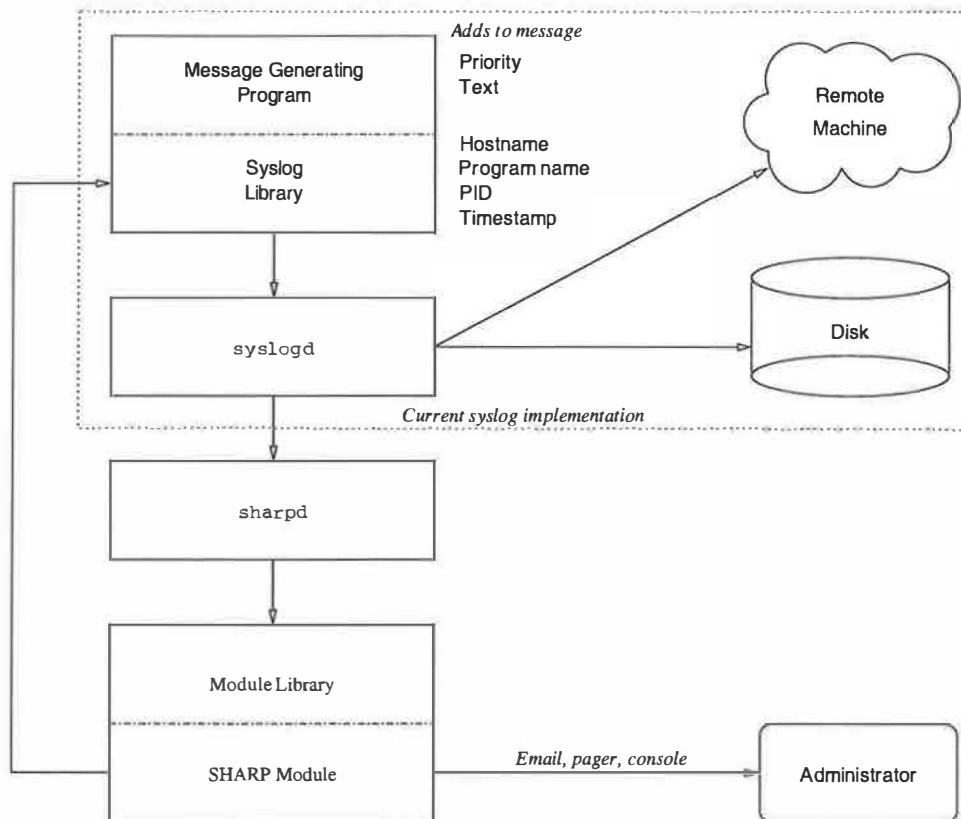


Figure 1: syslog and SHARP

the most prominent, read syslog messages in real-time, perform string based parsing, and direct the output. The limitation of this type of system is the ability to form complex routines based on messages. While you can configure this device to pass messages to an outside program, state is not saved across invocations. Assume an administrator has a swatch filter that dials a pager each time the company database is filled. When a database fills, it usually complains repeatedly, not just with just a single message. With a swatch style filter, the administrator will be paged as many times as the database error message occurs.

SHARP

All of these shortcomings with syslog point to the need for another tool that is able to perform real-time analysis and execute complex heuristics while retaining compatibility with the current style of system logging. The SHARP (Syslog Heuristic Analysis and Response Program) system attempts to address many of these architectural shortcomings. SHARP offers an interface for heuristic analysis programs to wisely utilize system logs. System administrators can decrease the burden of manually reviewing logs and improve the overall fluidity of the system by using SHARP.

Architecture

SHARP was designed for both simplicity and reliability. Simplicity is necessary to not obfuscate an already complex architecture of UNIX messaging. Reliability is necessary to guarantee the proper handling of messages and the overall stability of the system. A single mishandled message could lose information crucial to the administrator.

The SHARP architecture consists of three parts. SHARP modules are registered processes that perform some sort of heuristic analysis. The modules use a simple library interface to connect to `sharpd`, the SHARP daemon. `sharpd` communicates directly with the syslog system and receives a copy of every system message from `syslogd`. The library does the work of connecting to `sharpd` and passing the received messages back to the module.

Using the library interface, resident SHARP modules can specify the priority of messages they would like to receive. When `syslogd` receives a message, it passes it to `sharpd`, which in turn passes the message to only the interested modules. The modules perform their heuristics and can take various actions such as logging, emailing, or notifying the administrator.

When `sharpd` passes a message to a module, the message has been parsed and placed in a predefined structure to standardize analysis. This structure contains the time the message was received, the priority, the host that generated the message, and the text of the message. Communication with modules occurs over UNIX domain sockets. Using UNIX domain sockets provides further extensibility for replacement with

internet sockets. A SHARP system could forward messages to modules residing on separate systems for analysis.

To report its findings, a SHARP module may choose to send a syslog message. If a module reaches a conclusion based upon previous messages, this new syslog message could report its findings at a higher level than the previous messages. This higher level message might be noticed by the administrator, while the previous messages of a lower level may have slipped past. Modules could use syslog messages as a rudimentary version of interprocess communication. Care must be taken by the module programmer not to generate messages that would travel in an infinite loop between the SHARP and the syslog architectures.

Example Usage

For example, a module could track user patterns and report anomalous behavior. There are multiple user entry points into a UNIX system: `ssh`, `telnet`, `ftp`, `rlogin`, and so forth. Since each point of access is equally valid, it would be insufficient to monitor only one. SHARP is a perfect system to correlate this login information. The SHARP module would indicate to `sharpd` that it only wishes to receive messages about logins. Over time, the module would build up a database of user patterns such as the time of login and the remote host from which the user is accessing the system. When the module receives information about a new login, it would report anomalous behavior based on a configurable degree of confidence. Suppose a user normally only logs in to the system from a machine local to the office and only during business hours. When the SHARP module sees the same user logging in from a system across the Internet at 3am, it would report the strange behavior.

A SHARP module could be written to handle the problem of an administrator being inundated with pages due to a flood of messages. This SHARP module would maintain a queue of incoming messages to be paged, and throttle similar messages. With this module, the administrator could fine tune the ability of programs to flood the notification path of urgent messages.

A Complete System: `nsyslogd` and SHARP

As previously shown, the current syslog system is inadequate for a system such as SHARP to operate at full potential. A replacement for `syslogd` that does not have as many problems can be used. `nsyslogd` solves many of these problems by extending the functionality of the standard syslog system [5].

`nsyslogd` uses TCP instead of UDP for network message delivery. TCP, a connection oriented, reliable protocol, is much safer to use on long network paths. TCP is more difficult to spoof than UDP, but not impossible. `nsyslogd` can also use SSL over TCP, which gives network connections host authentication and data encryption.

nsyslogd uses a hash based algorithm [3] to guarantee message integrity. For each file destination a sister log is created containing chained hashes of the written logs. An external program is executed to check the integrity of the logs against the hash file.

Other syslog replacements have similar features, but nsyslogd has one prevailing feature that is useful to the operation of SHARP: priority preservation. When nsyslogd writes a message to a destination, it includes the priority information. SHARP uses this feature to filter messages for delivery to interested modules only. Without this feature SHARP is forced to flood each message to every module, which in turn is expected to do its own parsing.

sharpd communicates with nsyslogd by means of a UNIX domain socket. nsyslogd can be configured to write all system messages to a socket where sharpd receives them and passes them to interested modules.

nsyslogd was written by Darren Reed and is freely available on the Internet. It has been ported to multiple UNIX platforms. It has proven to be a reliable alternative to the standard syslog that, when paired with SHARP, becomes an even more powerful system.

Modules

Modules utilize the framework provided by SHARP. A module is defined as any program that utilizes the SHARP module interface and receives messages from sharpd. Developers of modules include a single header file and link their module against the SHARP library. The interface is presented along with a simple example that illustrates proper usage.

Module interface

The interface to the module library consists of only three core functions and is designed to be as simple and flexible as possible for the programmer.

`void sharp_filter (const char *fac, const char *lev)`

Register interest in messages of facility `fac` and level `lev`. A wildcard facility named `all` is

available that matches every facility. This function may be called any number of times, but must be called before `sharp_init()`.

`int sharp_init (const char *name);`

Connect to sharpd and register with the corresponding name. 0 is returned on success, -1 on error. This function must be called before `sharp_run()`.

`void sharp_run (void (*callback)());`

Begin receiving messages from sharpd. `callback()` is the function that handles a received message and must be of a particular prototype. `sharp_run()` is distinct from `sharp_init()` to allow for future developments in SHARP that perform run-time checks on modules when they connect to sharpd.

`void callback (sharp_msg m);` ---begin:quotation--->

The function defined by the programmer that performs analysis on the message passed as a parameter. When `callback()` returns, the `sharp_run()` function again waits for another message from sharpd.

The data structure containing the message from sharpd is of the following format:

```
typedef struct sharp_msg {
    time_t time; /* sharpd timestamp */
    char pri[]; /* priority */
    char host[]; /* originating host */
    char text[]; /* message text */
} sharp_msg;
```

Each string field is of a fixed length defined in the header file for security and robustness. Since this structure is just a copy of a message, the module is allowed to alter the contents of `sharp_msg`.

The function `atexit()` is used to force execution of a SHARP library function that deregisters itself with sharpd. The programmer does not explicitly have to deregister the module. Figure 2 shows the state of a module as it connects to sharpd, receives messages, processes them, and finally exits.

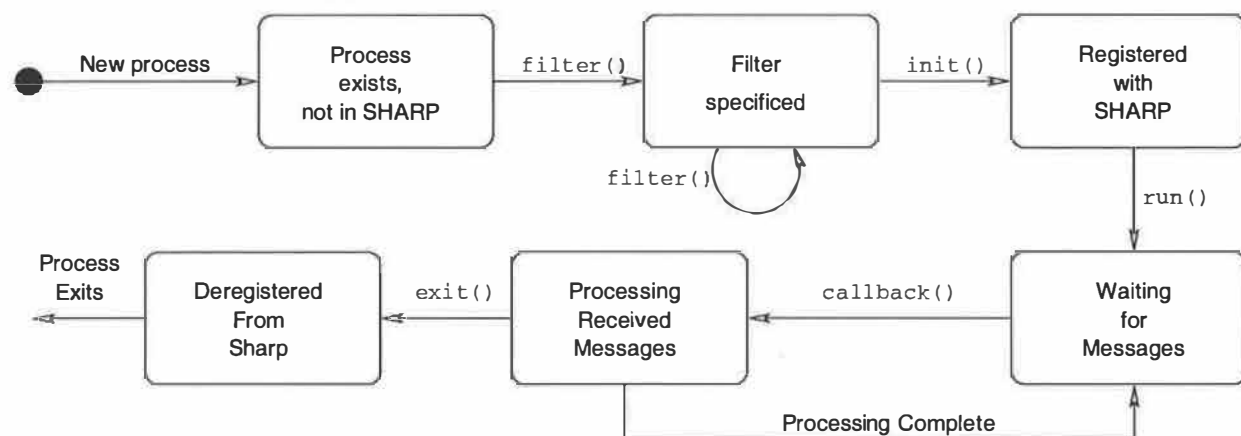


Figure 2: SHARP module state diagram.

Example Module: mark

To fully illustrate the simplicity and flexibility involved with authoring a SHARP module, an example is presented here to introduce the interface to potential programmers. This module is called mark because it reads syslog mark messages and reacts when a problem occurs.

syslogd can be configured to generate a message of facility mark at a predefined interval. This message is a "heartbeat" to show that syslogd is still running. The mark module reads these messages and reacts if it does not receive one at a predefined interval. Listed below is the source code, edited for both clarity and brevity. See Listing 1.

Note that this example is a skeleton and does require extra coding to be fully functional. The example only shows the code necessary to monitor syslogd on a single host. Adding support to the module for multiple hosts is straight forward.

Also included with the SHARP system are utility functions that modules would often have to use, such as sharp_email(), shown in the above example.

Current and Planned Modules

The authors have implemented and tested several SHARP modules. It is hoped that the SHARP web site

comes to be a central point for module sharing in the spirit of the Open Source movement. The modules which we have implemented or anticipate implementing include:

mark The mark module outlined above receives heartbeat messages from any number of syslog daemons on remote machines. If a message is not received in a configurable amount of time, the mark module assumes the remote host or network is down and informs the administrator.

userpattern A module that gathers statistical data of the system use of users and reports anomalies. For instance, a particular user is known to only use the system during office hours and connects only from a particular host. When the module notices that the user is online at 3am from an off-site host, the module would report this anomaly. This shows the power of a heuristic, stateful approach in monitoring.

useralert A module that alerts the administrator when a particular user connects to the system. This module can also be used to centralize login records for multiple machines.

dailyfilter This module emulates the functionality of syslog utility programs. It filters out all syslog messages based upon regular expressions and email the output.

```
#include "sharp.h"
#define TIMEOUT 60          /* threshold seconds */
static time_t last;
void main(int argc, char *argv[])
{
    sharp_filter("mark", "info");
    sharp_init("mark");
    signal(SIGALRM, alarm);
    alarm(TIMEOUT);
    sharp_run( (void*) mark_callback);
}

void mark_callback(sharp_msg m) {
    if(m.time-last > TIMEOUT)
        react();
    else
        last=now;
}

void alarm() {
    /* Did not receive the mark message */
    react();
}

void react() {
    /*
     * Possible reactionary methods include sending an email,
     * dialing a pager, or sending a syslog message.
     */
    sharp_email("admin@localhost", "syslogd died");
}
```

Listing 1: mark module.

problemalert Critical errors repeated more than a threshold number of times over a set period escalate notification (ie paging the sysadmin). Works well with the "all" wildcard.
 firewall This module receives firewall logs and analyzes trends and anomalies.

Experience using SHARP

The Computer Science and Information Systems department at Grand Valley State University maintains a network of machines known as the Experimental Operating Systems Laboratory, or EOS lab. The EOS lab consists of 28 Linux workstations and one central file and authentication server. Because of its classical network architecture, this environment was optimal for testing a working implementation of SHARP. Due to the enormous amount of system messages, they are generally ignored by the EOS administrators and only used as an autopsy instrument.

Results

It was determined each workstation generates an average of 250 system messages per day. The main server for the lab generates an average of 36,000 syslog messages per day. The total number of system messages generated in the EOS lab averages just above 42,000 per day. If this rate were normally distributed over the course of 24 hours, a message would appear every 2 seconds, way beyond the sensory capacities of any human to manually monitor. These statistics are highly site and service dependent, but should offer a general idea as to message rates.

The SHARP system has worked smoothly on this network. The concept of a centralized logging system seems to be the best operating environment for SHARP. Individual host installations of SHARP do not scale well when dealing with large networks. It is much easier to configure a new system to forward all syslog messages to a central logging system than to install SHARP and the various corresponding modules.

The EOS lab is a very small network in comparison to larger installations. SHARP is expected to scale well to these much larger systems. Due to its nature, the architecture is well suited for scalability. The centralized logging host could forward messages to instances of SHARP on different systems. Each system could have a unique set of modules. The socket architecture built into SHARP allows seamless changes between a local ==>[ignored: sc]<== UNIX-domain socket and a network socket. SHARP would thus easily support applications such as distributed anomaly detection.

Future Work

With the experiences gained from using SHARP, the authors have noted a few features that would both aid module developers and improve overall performance of the system.

- Patches for other implementations of syslogd to pass priority information with messages. SHARP users will not be forced to use nsyslogd.
- A global configuration file, such as /etc/sharp.conf, would define variables that modules could use. For instance, a module would call sharp_get_var(ADMIN) to get the email address of the administrator defined in the configuration file. This way potentially dynamic information would not have to be hardcoded into modules.
- Support for other languages besides C that have better text processing capabilities. Ideally a Perl module will be the first to be implemented.
- Support threadSHARP library calls.

Conclusion

The traditional UNIX message facility is inadequate for the active monitoring and response required of system administrators of even moderately large networks. Working with the existing syslog infrastructure, SHARP can improve the quality of monitoring, while at the same time reducing the burden on the system administrator.

Our goals for the design of SHARP have been met. SHARP is compatible with syslog, and hence requires no changes to future or existing message generating programs. SHARP's architecture is simple and clean, promoting extensibility and sharing of modules. The relatively small (approximately 2000 lines of code) implementation of SHARP is indicative of its simple and clean design.

SHARP's modules allow for continuous monitoring and the maintenance of state, which in turn support the implementation of monitoring heuristics not possible with other extensions to syslog. The modules we have implemented work as expected and have proven their utility in a production environment.

We have tested SHARP in a production environment of a network of Linux workstations and a single file/authentication server. Our testing has shown SHARP to improve the monitoring of a network of hosts by supporting a pro-active form of monitoring by the system administrator.

Availability

SHARP and a few packaged modules are available in source code form from the official web site: <http://www.csis.gvsu.edu/sharp>. This code is available under the BSD license [4]. Both the SHARP daemon and the modules have been compiled and tested on multiple platforms.

Author Information

Matt Bing will receive his M.S. in Computer Science from Grand Valley State University the day after the conference. He currently resides in Ann Arbor, Michigan working on intrusion detection systems for Anzen Computing. His email address is matt@csis.gvsu.edu .

Carl Erickson is an associate professor in the CSIS department at Grand Valley State. Last year he succumbed to start-up fever and is currently on leave working as a software architect with XiphNet, Inc. Reach him via email at erickson@csis.gvsu.edu.

References

- [1] Stephen E. Hansen, E. Todd Atkins, "Centralized System Monitoring With Swatchp" *LISA*, 1993.
- [2] "Security In Network Event Logging", August, 2000 IETF Draft available at <http://www.mail-archive.com/syslog-sec@employees.org/msg00466.html>.
- [3] Bruce Schneier, "Secure Audit Logs to Support Computer Forensics," *ACM Transaction on Information and System Security*, v.1, n.3, 1999.
- [4] BSD License <http://www.freebsd.org/copyright/license.html>.
- [5] Darren Reed, nsyslog, <http://cheops.anu.edu.au/avalon/nsyslog.html>.
- [6] FreeBSD /etc/security, <http://www.freebsd.org/cgi/cvsweb.cgi/src/etc/security>.
- [7] BSD source lib/libc/gen/syslog.c.
- [8] BSD source usr.sbin/syslogd/syslogd.c.
- [9] BSD source sys/sys/syslog.h.

Peep (The Network Auralizer): Monitoring Your Network With Sound

Michael Gilfix & Prof. Alva Couch – Tufts University

ABSTRACT

Activities in complex networks are often both too important to ignore and too tedious to watch. We created a network monitoring system, Peep, that replaces visual monitoring with a sonic ‘ecology’ of natural sounds, where each kind of sound represents a specific kind of network event. This system combines network state information from multiple data sources, by mixing audio signals into a single audio stream in real time. Using Peep, one can easily detect common network problems such as high load, excessive traffic, and email spam, by comparing sounds being played with those of a normally functioning network. This allows the system administrator to concentrate on more important things while monitoring the network via peripheral hearing.

This work was supported in part by a USENIX student software project grant.

Introduction

Are your systems and network functioning correctly? Can you be sure at this moment? Every administrator has some need to be able to answer these or similar questions on an ongoing basis.

Current approaches to live monitoring of network behavior (such as Swatch [10], mon [4], and their many relatives) can send email or page responsible people when things seem to go wrong. These tools are both visual and intrusive; operators must either be interrupted by alerts or periodically suspend other work to check on network status. Furthermore, these approaches are highly *problem-centered* and provide mainly *negative reinforcement*; the monitor notifies an operator only when problems occur. It does not, as a rule, regularly inform one when things are going well.

We created a tool Peep that represents the operational state of a system or network with a *sonic environment*. The flavor, texture, and frequency of sounds played are used to represent both proper and improper network performances, while the ‘feel’ of the sounds provides the listener with an approximation of network state. This environment plays in the background while the operator continues other tasks. Without looking anywhere and without interrupting other pressing activities, the operator can hear *peripherally* whether action is required.

Auralization

The idea of auralizing network behavior by playing network sounds is not new. Joan Francioni and Mark Brown [3, 5] represented parallel computer performance using a synthesizer driven by a MIDI interface. The strength of this approach, however, was also its main limitation. For music to remain pleasant, one must limit one’s representations to a limited number of relatively pleasing harmonic combinations. This greatly limits what one can represent with this technique. *Earcons* [2] are the sonic equivalent of icons;

sounds that are naturally associated with particular events. For example, most people associate a car horn with impatience or alert and a doorbell with someone entering a house.

Both of these approaches define the meanings of specific sounds or particular combinations in isolation. Combining sounds is difficult unless they are consonant either musically or environmentally, that is, that the sounds naturally occur together and ‘sound right’ in combination. Natural sounds have an advantage over music; they sound normal and pleasing in almost any combination similar to that of nature. For example, birds and frogs in wetlands can sing with virtually no coordination, and the result is still pleasing.

The Psychology of Audio Notification

What makes Peep possible is that events in networks have easily recognized natural sound counterparts. Moreover, numerous natural sounds can be played in combination while the result stays pleasing to the ear. If each sound represents some part of network function, and all are played together, the result is a *sonic ecology* in which the current state of the network can be determined moment by moment.

Peep exploits human instinct: our ability to notice a deviation from the norm with little effort, to determine what sounds right, and to discern singular important sounds from a collection of many sounds. We do these tasks with little or no conscious effort. Since computer interfaces mainly require the visual senses (and some motor skills), the audio senses are left available to perform this unconscious processing.

Furthermore, Peep takes advantage of our ability to do abstract processing. Instead of attempting the difficult and sensitive problem of determining when a network crisis has occurred or is about to occur, Peep provides contextual, continuous sound information and leaves interpretation to the listener. Decisions are based not only on the quantitative measure of things,

but the relative amount and absence of things. A musician friend has often expressed to me his philosophy: "Anybody can play drums, but the great drummer concentrates as much on the feel of the notes as on the space, or absence of sound, between them." Similarly, information that is lacking from Peep's sound ambiance is just as important as the amount of information conferred and the relative magnitude is left to the judgement of the listener.

Representational Techniques

Sound representation in Peep is divided into three basic categories: *Events* in networks are things that occur once, naturally represented by a single peep or chirp. Network *states* represent ongoing events by changing the type, volume, or stereo position of an ongoing background sound while *heartbeats* represent the existence or frequency of occurrence of an ongoing network state by playing a sound at varying intervals, such as by changing the frequency of cricket chirps.

Peep represents discrete events by playing a single natural sound every time the event occurs, such as a bird chirp or a woodpecker's peck. The sounds we chose are short and staccato in nature and easily distinguishable by the listener. Additionally, we noted that certain events tend to occur together and found it convenient to assign them complementary sounds. While monitoring incoming and outgoing email on our network, we noticed that the two events were often grouped together, since both types of email were usually transferred in a single session between mail servers. To better represent this coupling between incoming and outgoing email events and make the representation sound more natural, we used the sounds of two conversing birds. Thus, a flood of incoming and outgoing email sounds like a sequence of call and response, making the sound 'imagery' both more faithful to our network's behavior, as well as more pleasing to the ear.

State sounds correspond to measurements or weights describing the magnitude of something, such as the load average or the number of users on a given machine. Unlike events, which are only played when Peep is notified of them, Peep plays state information constantly and need only be signaled when state sounds should change. Peep represents a state with a continuous stream of background sounds, like a waterfall or wind. Each state is internally identified as a single number measurement, scaled to vary from extremely quiet to loud and obnoxious. Background sounds should be soothing while the network is functioning normally. However, when the administrator is annoyed, he will know that action is required.

Heartbeats are sounds that occur at constant intervals, analogous to crickets chirping at night. A common folk tale is that one can tell the temperature from the frequency of cricket chirps; likewise we can

represent network load as a similar function. Intermittent chirps might mean low load, while a chorus might mean high load. Heartbeats can also report results of an intermittent check (or ping) to see if a given machine, device, or server is functioning properly.

Humans are very apt at recognizing when continual background sounds change, making problem detection swift and simple. If your email server dies, chances are that you will not receive any email warning of the problem. But the crickets will have stopped chirping. The heartbeats provide an effective method for monitoring the functionality of your network and being alerted of a problem when all else fails, through the absence of sound. Likewise, the administrator need not fear about monitoring his Peep server; if it dies, he will be immersed in sudden silence!

Sound representation depends very much on personal taste. Peep aims to provide users with a choice of themes such as *wetlands* (the current theme available) or *jungle*. Within a theme, sounds are classified according to the network events they most appropriately express. Although the two chorusing birds were used to represent incoming and outgoing mail in the previous example, the two bird sounds could have been used for any type of coupled event behavior. These classifications help the user make decisions on what sounds to use from his collection of favorites.

We also recognize that distinguishing sounds can be difficult if, for example, several similar bird sounds are used in a single theme. As the theme repository provided with Peep expands, we hope it will address a wide range of network situations and personal tastes.

Scalability and Flexibility

The Peep architecture was designed to be versatile and scalable. The architecture is based upon a producer/consumer relationship between distributed monitoring processes that watch the network and servers that actually play sounds. Producers alert consumers to events and state changes via short UDP messages, as shown in Figure 1.

This architecture allows the receipt of status reports from any number of devices or nodes. Producers (the monitors in Figure 1) monitor network behavior and report events and states while consumers take their input from the producers and play the appropriate sounds. Producers can be pointed at several sound generators simultaneously, e.g., a lab full of Linux workstations, for a truly immersive experience!

Producers are executed as daemons on machines with access to information sources. This eliminates the need to send copious amounts of sensitive log or machine information across the network to a centralized monitoring server. The packets sent to the consumer contain only sound representation information and would be of little use to a snooper without access to the Peep configuration file.

The Peep system was designed to take advantage of existing system administration tools. Server and client configuration information is stored in the same configuration file. This allows centralized control of Peep via simple file distribution via NFS or other widely accepted mechanisms such as CFEngine [6, 7, 8] and rdist [9].

Clients provided with the Peep distribution are 'lightweight' Perl scripts. Each client functions strictly within one problem domain: it addresses its original intended purpose and no more. This keeps client code simple, easy to debug, and easy to customize.

We also wanted clients to run in the background and utilize as little resources as possible. Our log probing client, LogParser, watches log files and uses regular expressions to determine when particular events have occurred. Because of the way regular expressions are mapped in memory, scanning a single log for many different text patterns can become memory intensive. Instead, we designed LogParser to distribute monitoring overhead. Multiple instances of LogParser can run on separate feeds around the network, each instance searching for only a few textual patterns in the local system logs. This allows the system administrator to take advantage of the distributed computing power of his network, rather than waste what is often an abundance of idle resources in the hands of naive users. Peep aims to provide administrators with several means of implementing monitoring. Administrators still have the option of directing all log entries to a single machine should they so desire, at the cost of increased network bandwidth. Furthermore, the distributed method can be combined with the

single-machine method with no effort on the administrator's part.

Expanding the capabilities of Peep to fit your own needs is simple. Perl libraries handle all the low-level details, so writing scripts for event, state, and heartbeat-driven feeds can be quick and painless. LogParser can also be easily configured to scan a log for new events via additional regular expressions.

The Peep Protocol

Peep was designed to allow centralized management of its distributed architecture. The Peep protocol uses auto-discovery to dynamically bind clients and servers together upon startup. Peep configuration also uses a class mechanism to define groups of clients that should all report data to the same servers.

Peep was originally designed to use TCP for communication between clients and servers but communication over UDP proved much more efficient and effective. The main strength of TCP is its reliability. However, this reliability comes at the cost of greater bandwidth usage. Extra packets must be sent to ensure that transmissions were received correctly and in the proper order. Peep does not require packets to be reliable – since the representation of the state of the network is an approximation rather than a precise depiction. In any case, the human ear has no way of distinguishing the exact order of events when events rapidly arrive at the Peep server; indeed, the resulting sounds seem simultaneous.

The statelessness of UDP provided another benefit: clients and servers can be stopped and restarted

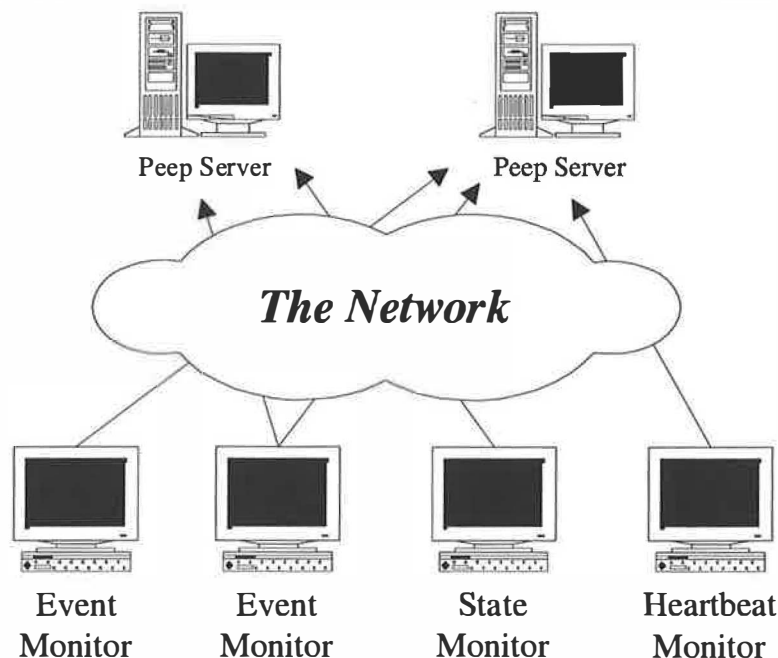


Figure 1: The Peep architecture.

without affecting one another. We wanted users to be able to write their own clients with minimal hassle. Avoiding connection management keeps clients simple and allows one to readily write Peep clients without making use of the included Perl libraries.

One drawback to using UDP is that clients have difficulty determining when servers crash. If this problem is not addressed, a client will continue to provide data to a non-existent server forever. Peep deals with this problem by combining a leasing mechanism with auto-discovery. This combination provides safe, dynamic, real-time bindings between clients and servers.

Peep's auto-discovery mechanism uses a domain-class concept to maintain bindings between clients and their respective servers. When a server initializes, it broadcasts its existence to the subnets associated with its classes and announces the classes of which it is a part. The clients that are members of those classes register themselves with the server and begin sending it packets. Conversely, should a client

start up and broadcast its existence, the servers associated with its class will tell it to begin sending. A broadcast only occurs once during the initialization of each client or server, after which a list of hosts is maintained on both sides and communications are direct. Both clients and servers can belong to multiple classes at the same time and clients can communicate with many servers concurrently.

Leasing is used to ensure that clients do not waste network bandwidth and system resources sending packets to servers that are no longer listening. The server sends a lease time to the client during auto-discovery. Just before the lease expires, the server tells the client to renew the lease. The client responds by telling the server that it is still alive and still needs to know about lease information. If the client has not heard from a server after the lease time has expired, it will no longer send packets to that server. Similarly, if a server does not receive lease acknowledgement from a client, it will no longer attempt to renew its lease with that client.

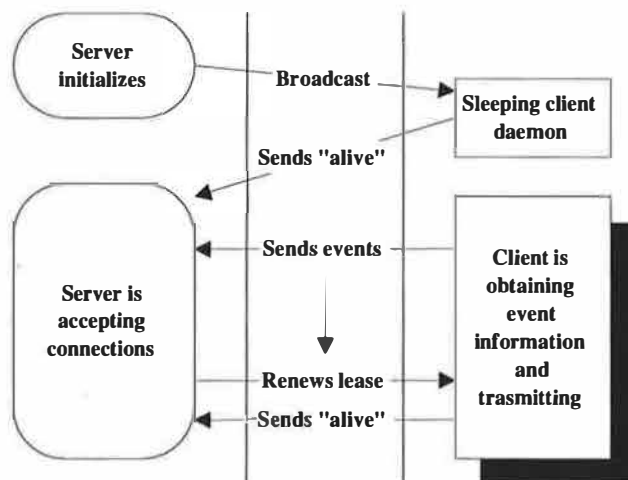


Figure 2: A server initialization.

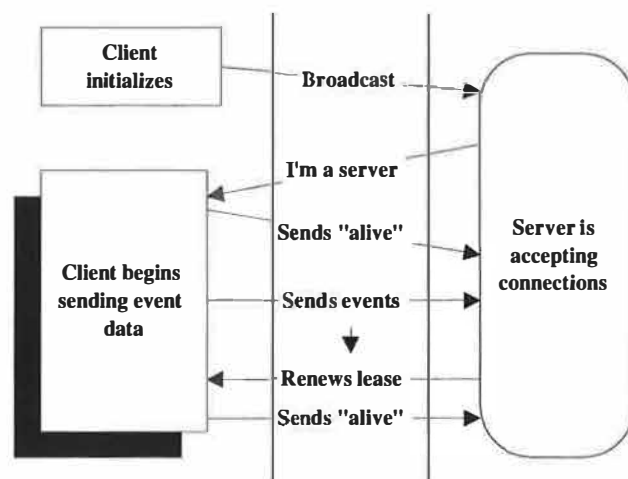


Figure 3: A client initialization.

The auto-discovery and lease mechanisms greatly ease the burden on the system administrator. The system administrator can then use a file distribution mechanism, like CFEngine, to add client and server daemons to a machine's background processes. Clients will sleep until a server becomes available, and will send packets only while that server stays available.

Alternatively, system administrators may decide to dedicate a machine to run Peep software and want all clients to execute on a single machine. In this situation, broadcasting becomes totally unnecessary and inefficient. Instead, the user can disable the auto-discovery mechanism. Clients will then become dumb clients, continually processing and sending event information to a server throughout the course of their lifetimes. Peep also provides the user the choice of mixing and matching, applying distributed and centralized configurations where they make sense.

In terms of robustness, the Peep protocol has version identification, room for future expansion, and type identification. Upgrades should allow older clients to work with newer servers and vice versa. Communications are done using one-byte quantities to represent attributes, and strings for anything more complex. This allows us to avoid any external data representation issues, making the protocol more portable.

Details of this protocol are hidden inside a Perl client interface library provided with Peep. The Peep library demands little expertise. To create a client with all of the library's benefits, programmers need only initialize the library with their application name and tell the library what information to send. Initializing the library parses the Peep master configuration file, so programmers need not do it themselves. This allows client design to be as simple or as complicated as the user desires. We hope that the simplicity of writing clients with the Perl library will encourage users to write their own client applications and share their code with others.

Configuring the Peep System

How one configures Peep is very much dependent on whether you choose to use single or multiple nodes. The generalized Peep installation is a four-step process: downloading the source and a sound package, compiling the server, editing the configuration file, and deploying clients.

The Peep server package uses the gnu autoconf package to make configuration and compilation easy. Support for tcp_wrappers [11] can be added as an option. Peep comes with two generic sound modules. One handles generic /dev/audio support while the other takes advantage of ALSA [1] on Linux systems. The configure package will default to ALSA drivers over generic support, if present. Special support for the Sun audio jack is also provided.

After compilation, the next step is to tell Peep which sounds to associate with which events, the classes to which your clients and servers belong, and your client configurations. A simple Peep configuration file is shown in Figure 4.

```
class myclass
  broadcast 130.64.23.255:2000
  server swami:2001
end myclass

client LogParser
  class myclass
    port 2000
    config
      #Name|OptLetter|Location|Priority|RegX
      out-mail 0 1 "sendmail.*:.*from"
      inc-mail I 255 0 "sendmail.*:.*to"
    end config
  end client LogParser

events
  #Event Type|Path|# sounds to load
  out-mail /path/sounds/peep1a.* 1
  inc-mail /path/sounds/peep2a.* 1
end events

states
  #Event Type|Path|# sounds| Fade time
  loadavg /path/sounds/water.* 5 0.3
end states
```

Figure 4: An example peep.conf.

Class definitions consist of two lines: one specifying broadcast zones and another specifying which servers are part of that class. Several broadcast zones and servers can be specified. Clients and servers can be part of several classes and will broadcast all the classes to which they belong during initialization. Putting multiple servers in a class (or making a client a member of multiple classes) is an easy way to have a single client dump data to multiple servers.

The 'events' and 'states' sections tell Peep servers to associate a name with a group of sounds. Filename descriptions in the Peep configuration file have a trailing asterisk extension followed by the number of sounds to load. Peep expands each asterisk into a two-digit number and loads, in ascending order, the number of sounds specified. All of the sound files loaded for a single entry then correspond to a single event. Every time that event occurs, the server will randomly play one of the associated sounds. This randomness makes the sound ambiance more natural. Heartbeats are created from streams of normal events from a client at suitable intervals. For state sounds, the server randomly strings together sound segments to create a non-repeating, random-sounding background ambiance. To keep transitions between sound segments sounding natural, the user can specify a linear fade time between segments.

The final step is to configure and deploy some of the clients provided with Peep. Two of those are discussed here: Peck and LogParser.

Peck

Peck is a command-line utility provided with Peep. It allows the user to tell a server to play (and how to play) a given sound. Peck is an example of a dumb client and bypasses the auto-discovery and leasing mechanisms. Event and state attributes are specified on the command-line and delivered directly to the server. Some command-line options apply to event sounds and others to background sounds, but the user need only remember a small number of options to get the Peep server to play some interesting things. Peck can be called with appropriate arguments from a shell script if a user does not wish to use a client library. Ideally, one should only utilize Peck to talk to servers on the same physical machine, or to report very infrequent events since Peck's inability to use auto-discovery and leasing capabilities means that calling applications will have no knowledge of the state of the receiving server. Peck is handy for a variety of simple tasks, including debugging installations, testing how things sound together, experimenting with Peep's capabilities, and interfacing Peep with other monitoring systems (such as an existing Swatch or mon installation).

LogParser

A simple log analyzer, similar to Swatch, is also provided with Peep. LogParser takes advantage of Peep's auto-discovery and leasing mechanisms. It is also an efficient distributed tool. LogParser reads its entire configuration but only searches for and remembers textual patterns specified on the command-line. It was designed to have multiple instances run on several different machines, each scanning for different sets of textual patterns on each client machine.

LogParser is flexible, easy to configure, and provides a simple way to access Peep's capabilities for representing events and states. It analyzes log messages as they are added to the log file and scans them for regular expressions. LogParser uses simple configuration syntax to generate command-line options and determine which sounds to associate with which particular events. Several options follow:

- The **priority** of the event ensures that no matter how many network events hit the Peep server, the most important ones will be played first and foremost.
- The **stereo location** of the event, aside from pleasing the true audiophile, helps the user distinguish and even locate an event. Sonic locations can even be assigned to correspond to the actual locations of machines on the network. Future versions of Peep might include a visual sound location map to exploit this.
- A **regular expression** that tells LogParser how to find the event in a log file. Users with experience with Awk/Perl pattern matching will appreciate this feature while others may find writing these difficult. We feel this is the easiest way to extend the capabilities of Peep without doing any sort of programming.

Directives in the LogParser configuration can be enabled or disabled via command-line options. Each line of the LogParser configuration corresponds to a user-specified single-letter option. In Figure 4, incoming and outgoing mail are mapped to command-line options "I" and "O", respectively. Thus, an invocation of LogParser searching for incoming mail might look as follows:

```
LogParser -events=I
          -logfile=/var/log/messages
```

Should the user forget the options, a help option will conveniently generate a list of user-configured options.

A single instance of LogParser can scan numerous logs simultaneously. It can send event streams to multiple servers automatically via the auto-discovery and domain-class mechanisms. These features provide the user with a myriad of options for structuring the architecture of Peep within a network.

Peep Performance under Pressure

To deal with copious amounts of incoming network data, Peep has a queuing and windowing system that handles large numbers of simultaneous events. This ensures that events are played in the order of receipt and in accordance with their particular priority. Peep will also discard events from its queue if too much time elapses between receipt and playtime, in order to keep events relevant.

Peep plays sounds by mixing sources in software. Since having large numbers of simultaneous voices can become computationally expensive, the user can tweak Peep's performance by changing the number of voices used when mixing sound. Less mixing voices tend to mean that the Peep's queuing and windowing system gets more usage, but the two always strike a balance to keep events accurately positioned in terms of time of occurrence.

It is difficult to send events to a Peep server fast enough to fill a queue on a Pentium II 400 and during testing, this required the use of an infinite loop. If the Peep server does manage to become overloaded, it only falls behind time-wise, adding a delay between the real-time event and the playing of its counterpart. Peep will preserve the general order and users will still be able to diagnose problems based upon the relative frequency of events. The delay experienced only applies to events and heartbeats; state changes occur instantaneously. In a worst case scenario, should the queue manage to fill up while new events are still arriving, Peep will begin discarding the oldest events from the queue, attempting to give the best approximation of network activity.

A Brief Overview of Implementation

The inner-workings of a Peep server are based upon the interactions between three execution threads as shown in Figure 5: the listener, the engine, and the

mixer. The listener handles all communications with the client, discovering clients via auto-discovery and keeping track of client leases. Upon receipt of event or state data, the listener thread places the information into a queue to be processed by the engine. The engine works closely in conjunction with the mixer to keep track of the priority of incoming and currently playing sounds. The engine also tries to find the best available mixing channel on which to play the incoming events and informs the mixer of the necessary parameters to properly represent the information. Should a suitable mixing channel not be found, the engine will place the events into a priority queue, ensuring that the mixer will play the most important events as soon as mixing channels free up. The mixer performs the processing necessary to produce Peep's output. This process involves scaling each sound's volume, as well as fading between state sounds. The mixer must also check the engine's event queue and ensure that queued, older events have priority as soon as mixing channels free up.

Critique

From our perspective, the design of Peep is very robust and portable. We decided, however, that support for generic audio hardware was more important than efficiency of memory and processor usage on the server side. Peep utilizes Linux ALSA and OSS drivers, as well as the Solaris /dev/audio interface, to avoid device incompatibilities. This is done at the expense of ignoring commonly available device-dependent hardware-based mixing in favor of mixing in software. Software mixing did afford us one advantage that hardware cannot guarantee: users will always get the benefit of sound processing incorporated into Peep regardless of the hardware. Future plans do include support for hardware-based mixing on a selected number of audio cards.

An invisible limitation of Peep is that creating accurate natural venues of consonant sounds is both an art and very labor-intensive. Due to copyright limitations on existing natural sound collections, Prof. Couch has spent many hours with a Telinga parabolic nature microphone and Sony DAT or digital minidisc recorder in search of the perfect bird. Sounds we collected required significant post-processing, including high and low-pass filtering and noise reduction, before they were free of enough normal background noises to serve as event sounds. Collecting state sounds proved even more difficult, with the sound of wind being the most difficult. The challenge was to collect 'desirable noise' without impurities such as car horns and airplane engines.

In spite of the excellent guidance on the recording of natural sounds that we obtained from the Cornell Ornithology website [13], the Stokes Field Guide to Bird Songs [14, 15], and the British Library National Sound Archive [12] we are not ornithologists and apologize in advance for any gross mislabeling of

sounds included with Peep! Nonetheless, we have made significant progress in providing a Wetlands venue, and are planning others in the future.

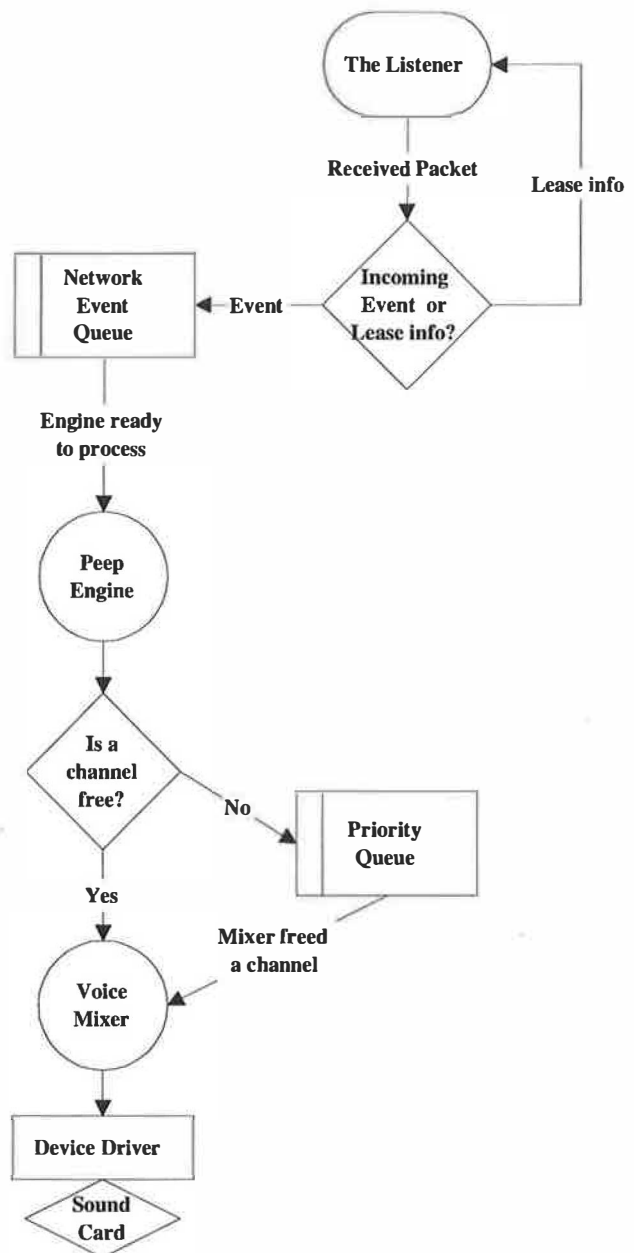


Figure 5: The Peep server's internal structure.

Configuring a Peep theme pleasingly can be non-trivial, especially when choosing which sounds should be associated with which events. The process of choosing sounds can often be a very lengthy. Since sounds chosen vary according to personal taste and the situation they are attempting to describe, we hope to provide several different preset configurations for our users after the tool has had more exposure.

Peep is relatively young and prior to this publication has received very little public usage. We hope we have anticipated and met the needs of a wide range of

network implementations. However, only public usage and time will tell.

Future Work

We want to see several other capabilities added to Peep servers to better represent network events. One idea is 'log dithering'. Due to block buffering, many log files are updated in erratic bursts so that several events are written to the log file and reported by LogParser as simultaneous. A dither time would space out how the events are played so they have a truer representation.

We also want to represent state sounds in a way that better models the way the human ear works. Since the ear hears amplitudes on an exponential scale (in dB), we want to scale state measurements exponentially so that they better approximate what the human ear considers truly loud. This still may not satisfy our vision of having a storm break loose when a machine is overloaded.

We may also allow sounds to change in nature with volume. A small stream might become a river rapid when a state measurement, such as load average, increases. State sounds might be represented by three or four different collections of sounds to achieve a 'thunderous' effect. A final item on the server wish-list is pitch bending: the ability to play sounds at different frequencies. Using this capability we could generate birdcalls at different pitches and then combine them together to create the effect of a chorus of distinct birds from a single sample.

We would also like to add a GUI to ease the process of configuring sounds for Peep. Since we plan on having several different sound classifications, a sound browser would be a welcome addition. The interface would let the user play several sounds simultaneously so they could get a feel for how things would sound in various situations. This will most likely be the next major addition to the Peep software package.

Lastly, we hope a few brave users will contribute homegrown scripts and configurations to the project so that we can establish an archive and ease the process of making a new installation.

Conclusions

This work began two years ago by trying to define what constitutes 'normal' behavior of a network and how to take action to rectify 'abnormal' behavior. This proved infeasible because normalcy depends as much upon policy decisions as upon many pre-existing conditions. These conditions exhibit complexities and intricacies that are difficult to depict via traditional methods.

Our sound ecology depicts normalcy in a new way. Things are normal when Peep "sounds like it did yesterday," regardless of the intricacy of the depiction. Our innate human abilities to detect these

differences are more acute than one may realize. When things sound different, we may not know why, but we can tell that *something* has changed.

Traditional tools look for specific problems while Peep only tells the listener about potential problems. In that respect, Peep will outlast traditional problem-detection tools because it portrays the general problem and no more. And unlike other tools, Peep is non-intrusive. One doesn't need to pay much attention to Peep in order to benefit. We don't want you to. We just want you to sit back, and listen.

Availability

The current revision of Peep is 0.3.0alpha and is currently freely available from <http://www.eecs.tufts.edu/peep/>. A demo of Peep's capabilities will also be provided on the website in .wav format so users can know what they're getting into before they install it.

Acknowledgements

Thanks to USENIX for funding this project and making it possible. Additional thanks goes to Andy Davidoff for contributing many great design ideas throughout the course of Peep's development and for being one of the first to embrace Peep software.

Biography

Michael Gilfix was born in Winnipeg, Canada and presently resides in Montreal, Canada where he attended high school at Lower Canada College. He is currently a junior at Tufts University, where he is completing his undergraduate degree in electrical engineering and his masters in computer science. His interests include guitars, music, and computers in all ways, shapes, and forms. While completing his degrees, he is currently practicing the art of system administration in Tufts' Electrical Engineering and Computer Science department. He will be graduating in 2003. He can be reached via electronic mail as mgilfix@eecs.tufts.edu. Reach him telephonically at +1 617-627-2804.

Alva L. Couch was born in Winston-Salem, North Carolina where he attended the North Carolina School of the Arts as a high school major in bassoon and contrabassoon performance. He received an S.B. in Architecture from M.I.T. in 1978, after which he worked for four years as a systems analyst and administrator at Harvard Medical School. Returning to school, he received an M.S. in Mathematics from Tufts in 1987, and a Ph.D. in Mathematics from Tufts in 1988. He became a member of the faculty of Tufts Department of Computer Science in the fall of 1988, and is currently an Associate Professor of Electrical Engineering and Computer Science at Tufts. He can be reached by surface mail at the Department of Electrical Engineering and Computer Science, 161 College Avenue, Tufts University, Medford, MA 02155. He

can be reached via electronic mail as couch@eecs.tufts.edu. His work phone is +1 617-627-3674.

References

- [1] Advanced Linux Sound Architecture, <http://www.alsa-project.org>.
- [2] G. Kramer, Ed, *Auditory Display: Sonification, Audification, and Auditory Interfaces*, Addison-Wesley, Inc. 1994.
- [3] J. Francioni and J. A. Jackson, "Breaking the Silence: Auralization of Parallel Program Behavior," *Journal of Parallel and Distributed Computing*, June 1993.
- [4] J. Trocki, "Mon, the Server Monitoring Daemon," <http://www.kernel.org/software/mon>.
- [5] M. Brown, "An Introduction to Zeus: Audiovisualization of Some Elementary Sorting Algorithms," *CHI '92 proceedings*, Addison-Wesley, Inc. 1992.
- [6] M. Burgess, "A Site Configuration Engine," *Computing Systems*, 1995.
- [7] M. Burgess, "A Distributed Resource Administration Using Cfengine," *Software: Practice and Experience*, 1997.
- [8] M. Burgess, "Computer Immunology," *Proceedings LISA XII*, Usenix Assoc., 1998.
- [9] M. Cooper, "Overhauling Rdist for the '90's," *Proceedings LISA VI*, Usenix Assoc., 1992.
- [10] S. Hansen and T. Atkins, "Centralized System Monitoring With Swatch," *Proceedings LISA VII*, Usenix Assoc., 1993.
- [11] W. Venema, "TCP WRAPPER, network monitoring, access control, and booby traps," *UNIX Security Symposium III*, September 1992.
- [12] "The British Library National Sound Archive," <http://www.bl.uk/collections/sound-archive>, The British Library, 2000.
- [13] "The Library of Natural Sounds," <http://birds.cornell.edu/lns/>, Cornell Lab of Ornithology, 2000.
- [14] D. Stokes, L. Stokes, and L. Elliot, *Stokes Field Guide to Bird Songs: Eastern Region* (three audio CD's), Warner Books, Inc., 1997.
- [15] Peterson Field Guides, *Eastern/Central Bird Songs* (three audio CD's), Houghton-Mifflin, Inc., 1999.

Thresh – a Data-Directed SNMP Threshold Poller

John Sellens – Certainty Solutions Inc.

ABSTRACT

Thresh is a simple SNMP [1] monitor, written in Scotty [2] (Tcl [3] with the Tnm extensions), which uses the UNIX file system hierarchy for configuration and data storage. Thresh compares SNMP variables to per-device thresholds or values, and issues notifications if any current SNMP variable values are unacceptable (or unexpected). Thresh can be used by itself, or as a complement to other network management and monitoring tools. Thresh can be thought of as fitting in between tools that generate immediate emergency alerts (such as Big Brother [4]) and trending and history tools (such as Cricket [5]).

Introduction

Virtually every computing system and network has some kind of monitoring mechanism in place these days, but in some cases the monitoring system consists only of users phoning and asking if there is something wrong with the network. For those interested in something a little more advanced or automatic, there are quite a few software packages available that do some form of monitoring. They range from the very simple (ping tests, etc.) to the very complex (large commercial packages that map networks, configure devices, and make your lunch), with various alternatives in between.

In the realm of “simple” monitoring software, packages can generally be divided into two types:

- Alarmers – Software that monitors connections, services, and so on, and sends out an alarm (mail, pager, smoke signal) as soon as it detects a problem. Some examples of alarmers are Big Brother [4], nocol [6], and Spong [7].
- Trenders and Trackers – Software that keeps history or trend data for future analysis or review. The most obvious examples here are MRTG [8] and Cricket [5].

A couple of years ago it occurred to me that there was another class of monitoring that didn't seem to be very well addressed – low to medium priority tracking of certain parameters and their values on computing systems and network devices. For example, you might want to track configuration changes, system or device reboots, or network interface status or change time. These are things that you may want to know, but which don't necessarily indicate an immediate problem and which may not be worth waking anyone up to investigate.

Thresh was created to provide this kind of monitoring. It tracks SNMP variables, compares them to threshold, pre-set, or last-observed values, and reports (typically via email) unexpected changes or out of range values.

Why Use Thresh and Not Something Else

I've become convinced that, ugly as it may sometimes seem, the Simple Network Management

Protocol (SNMP) [1] should be the basis for just about every monitoring system. Virtually every network connected device either comes with an SNMP agent or can run one with only a modest amount of effort. Most SNMP agents can provide a vast amount of (usually) useful information, and most agents for general purpose computers can be extended to provide just about any data that you might want to have.

Some monitoring systems (such as Big Brother and Spong) rely, to a greater or lesser extent, on separate client agents, running on each system that needs to be monitored, with a system-specific reporting protocol between the agent and the management station. This approach can be somewhat limiting (it's hard to use on things like networking equipment for example), can result in some duplication of services (if you need an SNMP agent for other purposes), and limits both what you can do (they're often not extensible), and where (as your firewall may not be able to pass the particular protocol implemented by the software). Thresh avoids these kinds of problems by using only SNMP for communication.¹

It's often useful to be able to track certain SNMP variables, but you don't always need to know about changes *immediately* – it's often good enough to hear about them the next time you read your mail. For example, the system.sysUpTime.0 SNMP variable gets reset every time a system or device gets rebooted – if you get notified every time that variable resets, you'll know if you've got a device reliability problem (or an extension cord that people keep tripping over). Similarly, you can generate disk capacity threshold warnings, network bandwidth warnings, network interface up/down notices, and so on.

With many monitoring systems, this kind of low-priority information can be hard to generate. Many of the most common freely available packages tend to have only a small number of notification or alert mechanisms, and are built with the idea that you're

¹SNMP suffers in some situations from being a UDP based protocol, but its advantages more than make up for its few disadvantages.

monitoring vital services and networks – sometimes everything is assumed to be an emergency.

Thresh was created to address this kind of medium-level monitoring need.

There are a number of commercial monitoring systems available. Some, such as Spectrum [9], HP OpenView [10], and NetCool [11], are widely deployed and very well respected, and most of them can do (or can be made to do) most or all of what thresh does. However, the commercial packages tend to require a much larger monitoring infrastructure, and a much larger commitment of time and money to implement, operate, and maintain. For small to medium sized sites, small, simple monitoring tools, like thresh, are often the best choice.

Implementation

Thresh was implemented in Tcl [3], using the Tnm network management extensions provided by the Scotty/Tkined [2] software. The Tnm extensions are a toolkit of Tcl procedures that make it easy to perform SNMP operations.

Tcl was chosen because it is well suited to this kind of task, and the Tnm extensions provided just the right functionality. At the time thresh was first contemplated, the SNMP modules for Perl were relatively primitive when compared to Tcl and Tnm.

Thresh has benefitted from the use of a scripting language, and the string and array manipulation routines provided by Tcl. Tcl allows the use of an interactive “shell” for testing and development. Being a traditionalist, I tended to develop incrementally, using the well-known edit, run, repeat cycle, rather than a more “modern” approach to program development. Thresh is currently about 700 lines of Tcl.

In retrospect, I’m still glad to have chosen Tcl in preference to Perl, C, awk, or Visual Basic.

Simplicity

One of the design and implementation goals for thresh was simplicity. That goal has been addressed in the following ways:

- One main program, thresh, and a very small number (currently one) of simple utility programs.
- Implemented in a well-designed scripting language (Tcl [3]), using a well-known and effective set of library routines (Scotty).
- Simple to run – requires no additional daemons (just a crontab entry), and no additional software is required on remote systems².
- Requires no special usersids or groups. (It would probably be useful to have a separate userid to run thresh, and perhaps a userid or group to

²Other than an SNMP agent which you should already have installed and configured anyway, and which likely came with your operating system.

own the config files, but that decision is left up to those installing and using the software.)

- Configuration is relatively straightforward, and involves only two simple file formats. The hierarchical configuration structure makes implementing default settings easy and flexible.

Data Direction and Configuration

Thresh’s configuration is “data-directed”³ – it is configured using a hierarchy of directories and configuration files that is intended to reflect organizational structure and DNS naming conventions.

The default configuration assumption is that each directory in the configuration hierarchy represents an element of the DNS name of the devices being monitored. For example, the sub-directory named com/whizbang/admin/printer1 would usually contain the thresh configuration for the device with the DNS name printer1.admin.whizbang.com. The name configuration directive makes it easy to override this default behaviour, by setting the DNS domain or node name associated with a particular directory.

Configuration Variables

Each directory may contain a DEFAULTS file, which sets the various configuration variables (such as name, notifier, delay, community, etc.) which control thresh’s behaviour. Thresh configuration variable names are case sensitive, and all include only lower case letters. Settings in a DEFAULTS file are in effect for that node and those lower in the hierarchy, unless overridden by a lower DEFAULTS file.⁴ Thresh configuration variables can also be set on the command line, in which case they override any other settings for the given variables. Figure 1 shows a sample DEFAULTS file.

```
verbose = true
name = mydomain.net
community = hello
mib = /usr/local/mibs/ascend.mib
# big network, long timeout
timeout = 20
notifier = threshmail jsellens
syslog = local1.info
```

Figure 1: A sample DEFAULTS file.

Thresh’s configuration variables include:

- walkonly – Walk the data hierarchy, listing the nodes and variable references found, but not querying, reporting, or logging.
- ignore – Defines a list of glob patterns of file and directory names to ignore. Setting

```
ignore = *
```

for example, ends up ignoring the data hierarchy below a given point.

³Or, at least, my definition of data-directed – I looked unsuccessfully for a more commonly accepted definition of the term.

⁴One could claim that this is similar to the class inheritance rules in object-oriented programming languages. Or not.

- **notifier** – The command line to run to send notification messages – the message text is provided as the standard input to the command. This can be set differently for different parts of a hierarchy, which makes it possible to assign responsibility for different nodes to different people or groups.
- **frequency** – The interval (in minutes) before otherwise identical notifications may be sent.
- **describe** – Whether or not to include a variable's MIB description field in notification messages. Description fields often end up being fairly generic, but including the description in messages can help make the notifications a little more self-documenting.
- **syslog** – A facility.level pair for logging messages to syslog.

The other configuration variables are described in the included *threshvars(5)* man page.

SNMP Variables

All other files in a directory are expected to contain a list of SNMP variables to monitor for that particular device, with comparison indicators and expected or threshold values. Thresh currently lacks a file inclusion mechanism, but the use of symbolic links makes it easier to manage the configurations for multiple, similar devices. A sample configuration file is shown in Figure 2.

The SNMP variable names used in a configuration file can be any string that Scotty will recognize as a particular MIB variable. They can be fully-qualified names, such as `iso.org.dod.internet.mgmt.mib-2.system.sysUpTime.0`, unique substings with common prefix elements removed, as in `system.sysUpTime.0`, or SNMP object identifiers (OIDs), such as `1.3.6.1.2.1.1.3.0`. OIDs aren't always the best choice, as they are typically somewhat more cryptic for the casual reader.

The first letter on each configuration line, C, G, I, L, S, or V, indicates the comparison to be made:

- **C**: Changeable – the variable's value may change, but should be reported each time it changes. This is useful for semi-static data, or

for monitoring things such as device interface status changes.

- **G**: Greater than – the variable is reported if its current value is less than or equal to the specified value.
- **I**: Increasing – the variable is reported if its current value is less than its previous value. This is handy for watching for reset times, such as the `system.sysUpTime.0` variable resetting when a device (or agent) restarts.
- **L**: Less than – the variable is reported if its current value is greater than or equal to the specified value.
- **S**: Static – the variable is reported if its current value is not exactly equal to the specified value. If no value is specified, then it is compared against the first-retrieved value of the variable. This is useful for monitoring things that should never (or almost never) change, such as `system.sysName.0`.
- **V**: Variable – the value can be anything, but it is queried and tracked to allow for later investigation or review.

The final field on some lines is the threshold value to compare against – a threshold value is required for G and L, optional for S, and not allowed for C, I and V. If a value for an S comparison is not provided in a configuration file, the first value for that SNMP variable retrieved from the device is saved and used as the “normal” value for the variable.

The configuration mechanism has proven to be quite flexible and easy enough to deal with, though some form of file inclusion mechanism would make some configurations simpler to create and maintain.

Notifications

Thresh provides a flexible mechanism for notifications. For each device described in the configuration hierarchy, if thresh determines that something needs to be reported, it formats a message and pipes it into whatever “notifier” program has been specified by the configuration variables. Thresh also keeps a copy of the message for internal reference when it is next

```
# this is a comment
S system.sysDescr.0
S system.sysContact.0
I system.sysUpTime.0
C interfaces.ifTable.ifEntry.ifDescr.5
C interfaces.ifTable.ifEntry.ifAdminStatus.5
C interfaces.ifTable.ifEntry.ifOperStatus.5
G ucDavis.memory.memTotalReal.0          90000
G enterprises.ucDavis.memory.memAvailReal.0 3000
L loadTable.laEntry.laLoad.1              1.20
L loadTable.laEntry.laLoad.2              1.50
L loadTable.laEntry.laLoad.3              2.00
V snmp.snmpInPkts.0
```

Figure 2: A sample configuration file.

run. If thresh observes the same problems (i.e., an identical notification message) the next time it is run, it will only send another notification if the specified frequency time has passed. If it sees a different set of problems when next run, it will report the complete current set of problems, regardless of whether or not the normal delay time has passed.

If the syslog variable is set, thresh will also generate syslog messages for SNMP variables that are out of range, in addition to the normal notifications. Messages formatted for syslog are deliberately terse, and include the node name, comparison type, SNMP variable, the normal or threshold value, and the current value.

Status, History and Logging

For each device in the hierarchy, thresh will create a configuration subdirectory named `.thresh`, as a convenient location to store the data it generates. Thresh maintains status, history and logging information, both for internal use and to make it possible to review past changes in state. It tracks the previous values of the SNMP variables (in `.thresh/status_*`), the last notification message sent (in `.thresh/last_complaint`), the date and time of last contact with the device (in `.thresh/last_response`), and a log of when variables were found to be outside their “normal” ranges (in `.thresh/log_*`).

The status and log files are named for the configuration files that they are related to, with a prefix of `status_` or `log_` added to the configuration file’s name.

Currently, there are no really interesting ways to access and use this data – you’re more or less stuck with using some paginator program to view the files. I should note that there is no built-in mechanism for log file rotation.

Integration with Other Systems

Thresh can be used effectively as a standalone, isolated monitoring tool, but it can also be integrated with other logging or reporting systems. Thresh’s core functionality is polling SNMP variables, comparing against pre-determined thresholds, and generating messages for distribution. Integration with other tools can be accomplished in two ways:

Syslog

Thresh can be configured to record out of range values to syslog, which provides an easy interface to any existing syslog watcher.

Custom Notifiers

By setting the notifier configuration variable, thresh’s alert messages can be trivially piped through arbitrary custom processes, that can record, mail, or dispatch as appropriate.

The message format is fairly consistent, simple, and relatively easy to parse. Additionally, the internal thresh code which actually generates the

messages would be easy to change if a specific output format was required.⁵

Scalability

Thresh is not arbitrarily scalable to huge numbers of devices and variables being monitored. Beyond a certain point, you will start to run into problems such as:

- Configuration complexity – a file and directory based configuration mechanism works fine up to a point, but beyond that you need configuration generators and a real database.
- No inherent parallelism – beyond some number of hosts and variables, you won’t have enough time to poll everything you want to poll within the time interval you would prefer. You can deal with this to some extent by running multiple parallel instances of thresh, but that adds administrative complexity.
- The default email notification mechanism can quickly get out of hand, if things start breaking. It is possible to use a smarter notifier, or to send notifications to different users or aliases based on location in the hierarchy, but that can get complicated. It would be possible to use a notifier that inserts messages into your management console, but if you have one of those, you may also already have more advanced commercial monitoring software.

Installation

Installation of thresh is very straightforward.

- You need relatively current versions of Tcl and Scotty (at least 2.1.x) on your monitoring host.
- Set the “hash-bang” line at the top of the thresh script to point to your Scotty installation.
- Install the script and man pages in the “usual” places.
- Set a cronjob to invoke thresh periodically, with a command line something like:

```
thresh topdir=/path/to/data
```
- And simply create a data hierarchy that describes the hosts, devices and variables that you wish to monitor.

The last step is admittedly somewhat more complicated and time-consuming than the others, but that is pretty much unavoidable. The distribution will include some sample configurations.

Enhancements

There are a number of enhancements to thresh that should probably be made:

- Thresh currently only supports SNMP V1 – it would be useful to enable all versions of SNMP supported by the Tnm extension.

⁵The `msgformat` configuration variable provides a rudimentary and somewhat unsophisticated mechanism for customizing the notification messages.

- Syslog logging is implemented through the external *logger(1)* command – it should be re-implemented using an internal library call.
- There should be some form of “include file” mechanism for configuration files, to reduce the need for the use of symbolic links to implement common configuration settings.

Lessons and Problems

Thresh seems to serve to illustrate a few useful lessons:

- Sometimes a variety of tools are needed to implement a complete monitoring “solution”.
- Simple tools are often quite useful.
- “Data-directed” design can be quite effective.
- SNMP can provide all sorts of useful information that can be difficult or impossible to obtain using other mechanisms.
- Tcl and the Tnm extensions are very useful tools.

Some of these also illustrate problems, the most obvious being that you probably need more than one monitoring tool, since no one tool is likely to do everything that you need done.

Generally, thresh has proven useful and fits nicely into an effective monitoring toolkit.

Availability

Thresh is “freely” available through <http://thresh.sourceforge.net/> or <http://www.generalconcept.com/resources/>.

Author Information

John Sellens is the General Manager for Certainty Solutions in Canada, based in Toronto. (Certainty Solutions was previously known as GNAC – Global Networking and Computing.) Prior to joining Certainty Solutions, he was Director of Network Engineering at UUNET Canada, and was a system administrator at the University of Waterloo for 11 years. He has a master’s degree in Computer Science from the University of Waterloo, is a Chartered Accountant, and is a semi-regular contributor to *login*. John, Joanne, and their delightful children live in Unionville, Ontario. Contact him at [<jsellens@certaintysolutions.com>](mailto:jsellens@certaintysolutions.com).

Bibliography

- [1] J. Case, M. Fedor, M. Schoffstall, and J. Davin, *A Simple Network Management Protocol (SNMP)*, Network Working Group, May 1990, RFC 1157, STD 15.
- [2] J. Schönwälder and H. Langendörfer, “Tcl Extensions for Network Management Applications,” in *Tcl/Tk Workshop*, pp. 279-288, USENIX and Unisys, Inc., Toronto, Canada, July 6-8, 1995.
- [3] John K. Ousterhout, “Tcl: An Embeddable Command Language,” in *USENIX Conference*

Proceedings, pp. 133-146, USENIX, Washington, D.C., January 22-26, 1990.

- [4] Sean MacGuire and Robert-Andre Croteau, *Big Brother FAQ*, <http://www.bb4.com/>.
- [5] Jeff R. Allen, “Driving by the Rear-View Mirror: Managing a Network with Cricket,” in *First Conference on Network Administration (NETA '99)*, pp. 1-10, USENIX, Santa Clara, California, April 7-10, 1999.
- [6] Vikas Aggarwal, *NOCOL – Network Operation Center On-Line*, <http://www.netplex-tech.com/software/nocol/>.
- [7] Stephen L. Johnson, *Spong – Systems and Network Monitoring*, <http://spong.sourceforge.net/>.
- [8] Tobias Oetiker, “MRTG – The Multi Router Traffic Grapher,” in *Twelfth Systems Administration Conference (LISA '98)*, p. 141, USENIX, Boston, Massachusetts, December 6-11, 1998.
- [9] Aprisma Technologies Inc., *Spectrum Network Monitoring and Management System*, <http://www.aprisma.com/>.
- [10] Hewlett-Packard Company, *OpenView Monitoring and Management Software*, <http://www.openview.hp.com/>.
- [11] Micromuse Inc., *Netcool Monitoring and Reporting Suite*, <http://www.micromuse.com/>.
- [12] University of California, Davis, *UCD-SNMP distribution*, <http://ucd-snmp.ucdavis.edu/>.

THRESH(1)

USER COMMANDS

THRESH(1)

NAME

thresh – a data-directed SNMP threshold poller

SYNOPSIS

thresh [*varname=value ...*]

DESCRIPTION

thresh is a data-directed SNMP threshold poller, and uses the file system for configuration, status, and logging. Each host or device to be monitored is configured in a separate directory, using files listing SNMP variables, values, and a comparison indicator.

In normal operation, **thresh** starts scanning a data hierarchy (as described in **theshdata**(5)) at a particular directory (set by the *topdir* variable), reading *DEFAULTS* files, variable files, querying hosts and devices, and recording and reporting the results.

thresh variables, as described in **threshvars**(5), and set in *DEFAULTS* files or on the command line, change **thresh**'s default behaviour and notification mechanisms. Any *varname=value* settings on the command line override both the built-in defaults and the settings in any *DEFAULTS* files encountered during processing.

thresh would typically be called periodically by **cron**(8).

EXAMPLES

In normal use:

% thresh

To use a non-default start directory:

% thresh topdir=/some/other/place

To traverse the data hierarchy and provide information on what would normally be queried:

% thresh walkonly=true

To do almost nothing:

% thresh 'ignore=*'

NOTE

thresh is written in **Tcl**(n), using **scotty**(1) and the **Tnm**(n) network management extensions.

FILES

thresh uses just about any file and directory names. The name *.thresh* is reserved for naming the subdirectories used by **thresh** to store status and logging information.

Any files matching *.thresh/log_** are log files, which will grow without bound, and which you should arrange to rotate, archive, or truncate periodically.

BUGS

thresh currently only works with SNMP V1. The logging to **syslog**(3) should be internalized in some way, and not depend on **logger**(1). There should be some mechanism for “including” one file from another, to reduce the dependance on symbolic links for sharing files. **thresh** is unlikely to scale to handle arbitrarily large networks.

SEE ALSO

threshdata(5)

threshvars(5)

scotty(1)

Tcl(n)

Tnm(n)

AUTHOR

John Sellens

NAME

threshdata – thresh data hierarchy description

DESCRIPTION

The **thresh**(1) SNMP poller uses a configuration hierarchy to direct its actions, maintain its status information, and store its logs.

Each directory in the configuration hierarchy (under the *topdir* directory) is assumed to relate to a network host or device, or to an intermediate name in a DNS naming hierarchy.

NAMING

By default, the *topdir* directory is assumed to refer to a device named “” – the empty string. Each directory below *topdir* normally adds one more element on the right hand end of a DNS name. For example, below *topdir*, the directory

org/usenix/conference

is related to the DNS sub-domain “conference.usenix.org”, and the directory

org/usenix/conference/ts1

is related to the device “ts1.conference.usenix.org”. This naming relation can be overridden by the use of the *name* variable.

FILES

Each directory may contain a *DEFAULTS* file, which contains variable settings (see **threshvars**(5)) that apply to that directory, and to all directories below that point, unless overridden on the command line or by other, lower, *DEFAULTS* files.

Any other files found in a directory (other than those ignored by the *baseignore* and *ignore* variables) are assumed to contain a list of SNMP variables to monitor. **thresh** uses sub-directories named *.thresh* to store status and log information.

thresh data files consist of zero or more lines in the following format:

```
<ws>type<ws>variable-or-OID<ws>value<ws>
<ws># comment ...
<ws>
```

where “<ws>” indicates white space.

The data file elements are defined as follows:

type	A single capital letter indicating the comparison to be made in determining “normal”.
C	Changeable – the variable’s value may change, but should be reported each time it changes. This is useful for semi-static data, or for monitoring things such as device interface status changes.
G	Greater than – the variable is reported if its current value is less than or equal to the specified value.
I	Increasing – the variable is reported if its current value is less than its previous value. This is handy for watching for reset times, such as the “system.sysUpTime.0” variable resetting when a device reboots.
L	Less than – the variable is reported if its current value is greater than or equal to the specified value.
S	Static – the variable is reported if its current value is not equal to the specified value. If no value is specified, then it is compared against the first-retrieved value of the variable. This is useful for monitoring things that should never change, such as “system.sysName.0”.
V	Variable – the value can be anything, but it is queried and tracked to allow for later investigation or review.

variable-or-OID

An SNMP variable name (or name fragment that **scotty**(1) can interpret) or SNMP OID (e.g. 1.3.6.1.2.1.1.3.0) to be monitored.

value A value for the comparison. Required for G and L, optional for S, and not allowed for C, I, and V.

SEE ALSO

thresh(1)

threshvars(5)

AUTHOR

John Sellens

THRESHMAIL (1)

USER COMMANDS

THRESHMAIL (1)

NAME

threshmail – mail notifier for thresh messages

SYNOPSIS

threshmail *recipient ...*

DESCRIPTION

threshmail expects notification messages from **thresh**(1) on its standard input, which it appropriately reformats into a mail message, and sends to every recipient given on the command line.

threshmail would typically be set as the *notifier* in a **thresh** *DEFAULTS* file.

AUTHOR

John Sellens

NAME

threshvars – configuration variables understood by thresh

DESCRIPTION

thresh(1) understands and observes certain configuration variables. Those variables can be provided on the command line or in files named *DEFAULTS* within the **thresh** data hierarchy.

Variable names are case sensitive and are expected to be in lower case letters.

DEFAULTS FILES

DEFAULTS files consist of zero or more lines in the following format:

```
<ws>VAR NAME<ws>=<ws>VALUE<ws>
<ws># comment ...
<ws>
```

where “<ws>” indicates white space. Values can contain embedded blanks.

Variables set by a *DEFAULTS* file apply at that level of the data hierarchy and below, unless overridden on the command line or in a *DEFAULTS* file further down the tree.

GENERAL VARIABLES

debug Generate debugging output.
Boolean. Default: true

verbose Generate informational messages.
Boolean. Default: true

walkonly
Walk the data tree, describing the hierarchy, but not querying, reporting, or logging.
Boolean. Default: false

topdir The top of the data hierarchy.
Default: /usr/local/thresh

name The DNS name or partial name of the device or hierarchy represented by the current directory in the data hierarchy. Gets extended by the name of each directory as **thresh** descends down the hierarchy, but can be overridden in a *DEFAULTS* file.
Default:

baseignore
The base list of “glob” patterns of file and directory names to ignore in the data hierarchy. If you override this variable, make sure that you end up ignoring the *.thresh* status directories.
Default: ... * CVS RCS README README.* *DEFAULTS* core *.core

ignore The extended list of “glob” patterns of file and directory names to ignore in the data hierarchy. Having two variables makes it easy to augment the default list of names to ignore without overriding the base list. Note that setting
ignore = *
will cause the hierarchy rooted at that location to be excluded from all processing.
Default:

prune If true, do not process further down this hierarchy if the current node is unreachable. This is essentially the equivalent of setting
ignore = *
if the current node is unreachable. This is useful, for example, for limiting the error messages that are generated if a gateway router is unreachable.
Boolean. Default: false

SNMP VARIABLES**community**

The SNMP V1 read community string to use to query hosts and devices.

Default: public

mib

Specifies a file name that contains an SNMP MIB that will immediately be read and compiled into the running program.

Default:

timeout

How long to wait for a response to an SNMP get request, in seconds.

Default: 10

retries

Number of times to retransmit an SNMP get request during the timeout interval.

Default: 5

NOTIFICATION VARIABLES**notifier**

Pipe notification messages to this program, often a mailer or mail interface.

Default: /bin/cat

frequency

Minimum time before sending another identical notification message, in minutes.

Default: 30

describe

Whether or not to include a variable's MIB description field in notification messages.

Boolean. Default: true

msgformat

A printf-style format string used to print notification messages, with the (cryptically named) variables smnpvar, message, complabel, compval, newlabel, newval, desc. This could use a little more sophistication.

Default: %s: %s\n %s %s\n %s %s%s\n

LOGGING VARIABLES**log**

Write out of spec entries to a log file.

Boolean. Default: true

logger

A command like **logger**(1) that writes to **syslog**(3).

Default: /usr/bin/logger

syslog

A syslog facility.level pair, as accepted by the **logger**(1) command, such as "local1.info". If set, out of spec entries are piped to the *logger* command.

Default:

syslogtag

Tag to use on syslog'd entries.

Default: thresh

SEE ALSO

thresh(1)

threshdata(5)

AUTHOR

John Sellens

eEMU: A Practical Tool and Language for System Monitoring and Event Management

Jarra Voleynik – eEMUconcept Pty Ltd

ABSTRACT

This paper describes a new monitoring and event management concept. eEMU is a client-server system that provides for rapid development of monitoring agents. This is thanks to its messaging language that takes advantage of heuristic algorithms implemented in the eEMU server. As opposed to SNMP and other static monitoring methodologies used in system and application monitoring, eEMU takes a whole new approach by incorporating the dynamic aspect of application and system events into the very heart of the dynamic message processing server.

Motivation

The society and industries are increasingly becoming dependent on computers. Consequently, system and application uptime management is becoming a critical issue. Performance of outsourcing companies is measured by SLAs (Service Level Agreements). With the advent of e-commerce on a global scale, ISPs are becoming 24x7 environments as well. It all calls for good, flexible and reliable monitoring tools that would allow to proactively keep uptimes at the highest level and at the same time measure performance of IT departments.

Monitoring tools must be simple, cross-platform, scalable and cost-effective. This, to my observation, is not the case with most available offerings. Utilities such as Big Brother are widespread since they are cost-effective open-source alternatives and they are relatively simple to use. Nevertheless, alarm presentation and agent technology in Big Brother does not bring any new major features compared to other expensive commercial offerings.

Enterprise offerings such as Tivoli, OpenView, BMC Patrol and Unicenter TNG use complex multi-level drill-down alarm representations. Multiple windows must be opened in order to identify the cause of an alarm. At the top level, alarms are usually represented with color-coded icons. Big Brother has borrowed this concept and offers color-coded button matrix for monitored resources with a drill-down option to find out more about some alarms. eEMU replaces color-coded icons with a simple intuitive interface called eEMU browser. Unlike other event viewers, the eEMU browser displays a single textual message for each event, not every event occurrence. Subsequent updates to the event merely update the existing message. As a result, a complete view of the enterprise is facilitated with a single message screen without a need to drill down for more information or consult the event log. By default, the eEMU browser displays only resources in "alarm state". It is based on

the premise that "normal state" information is in most cases of little practical benefit.

eEMU was designed with modularity and integration in mind. Through scripting hooks to the event engine, integration with other management tools, such as OpenView or Unicenter TNG, is simple to achieve.

eEMU was developed by Jarra and Anna Voleynik of eEMUconcept after many years of experience on large datacenter sites. Its design was an effort to overcome shortcomings of other monitoring offerings and provide a simple, powerful tool. Around 80% of monitoring needs are application or site specific. Standard monitoring agents that come with other vendor's offerings will not monitor applications using off-the-shelf agent configurations. A well monitored site incorporates plenty of customisations that should be simple to make. Here, eEMU comes to the rescue by providing a message type arsenal that greatly simplifies the design of monitoring agents. In the resource ID section I will touch on the flexibility with which eEMU can process resource hierarchies. To date, eEMU has been providing the simplest agent development kit available, thanks to the fact that the alarm status is maintained by the eEMU server rather than by the agent. Most eEMU agents are so simple and small that they have been coined micro-agents.

How eEMU works

The majority of the currently available tools are based on the premise that the agent "remembers" the state of resources it monitors. Consequently, agents are relatively complex programs that run as daemons and maintain either a status database or status files. eEMU works on the premise that all the status information is handled by the eEMU server. eEMU agents are simple scripts or programs that use the `emsg` program to send messages to the server. The `emsg` program is a simple executable that uses the eEMU protocol. The protocol is designed to deliver the necessary information to the server so that the server can

maintain the status of monitored resources. The key attributes of each message are resource ID, time-to-live, and message type. There are multiple message types that implement the eEMU messaging language.

The eEMU server stores current alarms in an in-memory database. Alarm messages sent by agents are matched against the database contents to find out if the status of an alarm has changed. If so, the alarm status is updated. Each status change triggers an action script. For the action script, message attributes are facilitated as environment variables that can be used to take conditional actions or perform correlation of messages.

SNMP monitoring systems are known to produce UDP storms. This is partly caused by the fact that SNMP traps are not guaranteed to be delivered, therefore the SNMP manager must regularly poll for resource status as well. eEMU has been tested to easily monitor 100 systems on a 33 Kbps dialup line. This is due to the fact that the OK status is not transmitted and there is no polling necessary by the eEMU server. eEMU uses the TCP reliable protocol and any message delivery problems are quickly revealed through missing heart beats from the heart beat agent, which is an important part of every eEMU agent kit. eEMU has been tested to process 1000 messages a minute on a

400 MHz Pentium PC. On a large site with hundreds of nodes it may be easier to deploy 2 or 3 tiers of monitoring servers with a consolidation server at the top. Messages from middle tier servers can be selectively forwarded to the higher level server. To make a reasonable effort of delivering messages, msgd uses an exponential back-off for spacing out several connection attempts before giving up and returning a failure code. Consequently, out-of-band communication channel can be used by agents if the primary communication channel is down.

The user interface is represented by an intuitive event console called eEMU browser. The browser unifies the best of both worlds - drill-down icons and scrolling event screens. To enhance user communication, the browser allows the operators to not only annotate existing messages but send new ones as well. Technically, there is no difference between messages sent by scripting agents and messages sent by operational staff. The whole purpose is to create an event flow between enterprise components and staff.

Figure 1 shows a list of alarms that are color-coded by severity for easy interpretation. The "user root is logged in" message is annotated with a change request number to indicate a scheduled work on the firewall.

Fl Num	Time	Source	Class	Sev	Message	Comment	Alias
-M 4449	14/03 09:27	osmi	UNIX/BACK	2	Backup.Osmi is running		
C- 4540	14/03 09:11	orange	UNIX/FW/USER	2	user root is logged in	changes 4235 Firewall	
-- 3161	14/03 05:37	apple	UNIX/FW/HAIL	2	mailqueue is 3102 long		Mail Gateway
-- 2104	14/03 02:58	miri	UNIX/OS/LOG/FILESIZE	2	/var/adm/messages is 5MB in size		Web Server
-- 678	14/03 07:16	oraga	ORA/PROD/PRO	2	oraga.archival process is down		Human Resources
-- 4344	14/03 00:39	dumbo	UNIX/OS/FS	3	/usr filesystem is 97% full		Print Server
-- 3874	14/03 07:24	kiwi	UNIX/PRO	3	sendmail process is down		Mail Server
-- 656	14/03 08:12	wickey	UNIX/HTB	3	heartbeat has not arrived		

Figure 1: Color coded alarms.

Alias	Fl Num	Time	Source	Class	Sev	Message	Comment
Firewall	C- 4540	14/03 09:11	orange	UNIX/FW/USER	2	user root is logged in	changes 4235 Firewall
Solution Test	0	1	0	0	0		
Human Resources	OR -- 678	14/03 07:16	oraga	ORA/PROD/PRO	2	oraga.archival process is down	
Development	0	0	0	0	0		

Figure 2: Business view mode.

Figure 2 shows the business view mode with alarms logically grouped for a consolidated functional view of the enterprise. Alarms under a specific business group can be displayed by clicking the business group button.

Resource ID and Message Class

In order to identify monitored resources, each resource is assigned an Object ID. For a simple Object ID processing, the Object ID must form the first word of each message. To uniquely identify a resource, we need a node name as well. Consequently, "node_name": "Object ID" form a Resource ID, which is a message key maintained by the eEMU server for every alarm. All references to existing alarms are done based on the Resource ID. Examples of Resource IDs are: web.dumbo.com.au:/usr or ps.porky.com.au:printer.sydney.accounts1.

If a new agent is written, its messages will be sent with its own resource ID. Therefore, the resource ID should fit into the existing resource ID hierarchy, just like SNMP OIDs are constructed based on a hierarchical principle. With eEMU there is no need to register the resource ID since the server doesn't need to know it in advance. It makes deployment of new agents and resource hierarchies an extremely simple task compared to SNMP agents. Today, systems and applications undergo a constant change, thus representing a very dynamic environment that calls for an equally dynamic monitoring tools.

If we start integrating eEMU with a call logging system, we will find that calls need to be logged once only for a specified resource problem. Later messages for that particular resource need to update the existing call only. Resource ID makes call logging possible.

Resource IDs are part of eEMU agents and can be part of the agent code or part of the agent configuration.

Time-to-Live

Time-to-live is a critical attribute of each message. It determines how long the message is kept in the status database maintained by the server. Let us demonstrate the time-to-live on a simple agent scenario. A filesystem agent runs every 5 minutes. It scans all the mounted filesystems and compares their utilisation with predefined thresholds. If a threshold is exceeded, a message to that effect is sent to the eEMU server. Time-to-live attached to the message will be set to 7 minutes. The server will store the message in the status database for the period of 7 minutes. If the message is not updated within 7 minutes, the server will delete it. The next filesystem agent run will occur 5 minutes later. It will send another message to the server if the offending filesystem's threshold is still exceeded. Since the previous message is still stored in the database (with 2 more minutes to live), the refresh message will re-set its life to 7 minutes. The situation

repeats itself as long as the filesystem's threshold is exceeded. Once the problem is fixed, the message times-out and is removed from the status database by the server.

Time-to-live concept allows to transfer alarm status management responsibilities onto the server, thus tremendously offloading eEMU monitoring agents. The concept of time-to-live can also be used in the reverse sense as implemented in the sleep message type, described below.

Message Class

The message class attribute was designed for flexible classification of resources, building business rules, event logging and escalation. The class can be used to hierarchically build classes or trees of resources and group them. The class attribute can decide which support group needs to know about the event and subsequently take action. Also, it may describe a resource hierarchy from the application or system point of view. Thus, class assists in both, the business escalation process (it adds a business layer to the message) and resource identification. Since the class attribute can be retrieved from within action scripts, it is trivial for administrators to implement conditional message processing based on the class string.

Message classes are part of eEMU agents and can be part of the agent code or part of the agent configuration.

Class examples are: /UNIX/PRD/FS, /NT/DEV/ORA.

eEMU Messaging Language

Analysis of monitoring agents behavior revealed that there are typical scenarios that repeat themselves. The findings prompted us to turn these scenarios into event message types, which resulted in the eEMU messaging language.

The eEMU messaging language and time-to-live are the key ingredients for making eEMU so simple to use. A fairly complex monitoring scenario can be expressed in a few lines of code.

Currently eEMU supports ten types of messages: normal, delete, count, sleep, wakeup, event, mask, query, comment and control.

The **normal** message is held in the status database for the time-to-live period, whereupon it is removed from the database if it is not refreshed. By setting time-to-live to slightly more than the agent polling interval, the message is guaranteed to stay in the status database for the duration of the alarm, thus reflecting alarm status.

The **delete** message can be sent to the server to delete a message.

The **count** message behaves similar to the normal message only it is in an inactive state until the

message has been refreshed a predefined number of times. Each count message is accompanied with a "lag" value that indicates how many successive alarm messages need to be received before the resource status is changed. As an example, a system administrator is not usually interested in CPU reaching 100% for short periods of time. It becomes a problem only if the CPU utilisation is high consistently over longer periods of time. There are many resources exhibiting similar qualities, e.g. the swap space, network traffic and memory utilisation. Count messages are a powerful means for monitoring such resources.

The **sleep** message is kept inactive (sleeping) in the status database. It becomes active when it times out. Of course, it can be removed from the database with a delete message before the actual time-out. The premature sleep message removal is used in many scenarios such as backups. When a backup is started, a sleep message is sent to the eEMU server. Time-to-live of the sleep message is slightly greater than the maximum duration of the backup. At the end of the backup, the sleep message is deleted. There are two conditions under which the backup raises an alarm. The first is that the backup script failed or crashed. The second is that the backup is running overtime. Both conditions will be revealed by the sleep message timing out and becoming active.

One of the most interesting uses of the sleep message is a heart beat agent that monitors system uptime. E.g. the heart beat agent on node "dumbo" keeps sending heart beat messages, such as

```
$ emsg -o sleep -n eemuserver \
  -p 1965 -t 7m -s 1 -c /UNIX/HB \
  -w icecream -m "heart-beat is down"
```

The above message is sent to "eemuserver" listening on port 1965, time-to-live of 7 minutes, severity 1, class /UNIX/HB, password "icecream" and message "heart-beat is down". The resource ID is "dumbo:heart-beat".

With heart beat agents, no node discovery is necessary. The first heart beat registers the node with the eEMU server. If the node is down, the heart beat message is not refreshed. The server turns the sleep message into an active message that triggers an action and displays the alarm on the eEMU browser. Once the problem is fixed, the message goes to sleep again by switching its status to inactive. After giving some thought to the heart beat concept one finds out that it is completely self-maintained. Frequent adding of new nodes to a large monitored base is as easy as sending the first heart beat without touching the eEMU server configuration. Also, removing a node is as simple as deleting the heart beat sleep message from the eEMU database. Experience from large scale eEMU deployments has shown that heart beat agents are an extremely reliable means of detecting problems.

The **wakeup** message is used to wake up a sleep message. Let us suppose that there is a sleep message

already in the database but we would like to expedite its time-out. It can be accomplished with a wakeup message.

The **event** message behaves just like the normal message with the difference that by default it does not show on the eEMU message browser. It is predominantly used to notify the eEMU server of an event and subsequently keep the event in the database for the event's duration. Event messages trigger action scripts and many times they are used just for that. At other times, a query message (described below) is used to query the status database for a particular event. E.g., a batch job on system A sends an event message to eEMU on successful completion. System B runs its batch job only on condition that the batch job on system A completed successfully. Before the batch job on system B runs, it first checks with a query message if an event that designates the system A batch job's successful completion exists.

The **mask** message instructs eEMU to mask out messages of a particular resource ID. The effect of the mask message is set for a time-to-live interval. It lends itself to scenarios such as when a backup shuts down an application that is being monitored. The application alarms can be masked out for the duration of the backup.

The **query** message has two main uses. The former use lies in querying the status database to find out about other alarms. This is handy for cross-system event correlation. The latter use lies in downloading files from the eEMU server. Central storage of files on the server can be used for software distribution purposes among others.

The **comment** message is used to annotate existing eEMU messages. IT staff can instantly provide information related to raised alarms, such as actions taken to correct existing alarms or reasons for scheduled downtimes.

The **control** message is used for administrative purposes such as suspending the eEMU server.

eEMU Agents

eEMU agents have the eEMU messaging language at their disposal. The core of the agent is the emsg executable. In many cases, the agent code is a few lines of code. This is due to the intelligence built into various message types eEMU makes available.

Most monitoring systems rely on thick agents that maintain status of monitored resources. The agent technology can be tapped into by using a provided SDK (Software Development Kit). Apart from a need to have programming skills, administrators are bound to learn the API and develop agents that are not portable, depend on one vendor and are hard to maintain. With eEMU, the agents are typically simple scripts in a language of the administrator's choice. Due to the agent's simplicity and scripting language

implementation, they are portable, developed in a rapid time-frame and easy to maintain. Moreover, eEMU agents can be simply scheduled through the UNIX cron or other scheduler, thus further simplifying the agent code.

To demonstrate eEMU features, below are a few scenarios and the way they can be tackled with eEMU.

A batch job run takes more than 10 minutes. If it is less than 10 minutes, the job terminated prematurely and we want to know about it. The following is the eEMU wrapper code:

```
emsg -n eemuserver -o "count 2" \
    -t 10m -c /VMS/BATCH \
    -m "batch_job failed"
start batch_job
emsg -n eemuserver -o "count 2" \
    -t -1 -C /VMS/BATCH \
    -m "batch_job failed"
```

Before the batch job is started, a count message with the lag value of 2 is sent to the server. This message will be sitting in the server in an inactive state for 10 minutes and unless it is refreshed, it will be deleted. Notice that the second emsg is sent for the same object ID, namely batch_job, whereby the lag of the count message is also set to 2. It means that if the first message is still in the database at the time the second message arrives, the message will assume an active state and displays in the eEMU browser.

A cold backup of an oracle database on system "porky" shuts the database down. Since the database processes are monitored with a process agent, we are inevitably going to receive an alarm that the database is down. In case of planned downtimes, we do not wish to receive alarm messages. Some monitoring systems offer a calendar feature that can be used to temporarily stop alarms from occurring. The problem is that a "static" calendar configuration needs to be attached to the alarm. With eEMU, message masking is dynamically adjusted to the oracle backup interval.

```
emsg -n eemuserver -h porky -o mask \
    -t 2h -c /PROD/ORA \
    -m "sap_oracle is down"
shutdown_oracle.sh
backup_oracle.sh
startup_oracle.sh
emsg -n eemuserver -o delete \
    -m "porky:sap_oracle"
```

The first emsg sends a mask message for the porky:sap_oracle resource ID. It will mask out sap_oracle alarm messages for 2 hours. Next, the database is shut down and a backup performed. After the backup has been completed, oracle is started up and the mask message deleted. A bonus is that if the backup runs overtime (more than 2 hours in this case), we will be alerted of the application being down beyond the backup window.

A CPU agent can be as small as the following single line:

```
eval `vmstat 1 3 | tail -1 | \
    awk '{if ($16 < 1) { print \
        "emsg -n eemuserver -p 1 965 \
        -c LINUX/CPU -t 7m -s 1 \
        -o \"count 5\" \
        -m \"CPU is 100% utilised\""}}`
```

The above line retrieves the CPU idle time from a vmstat output and if the idle time is less than 1% it sends a count eEMU message with a lag of 5. If five such messages are received in a row, an alarm is raised.

In order to expand the usability of eEMU, the query message was equipped with a file download option. As a result, a file can be downloaded from the eEMU server. This feature can be used for software or script distribution but also for ad-hoc remote job execution. The following simple task distribution agent regularly (from cron) checks if there is a specific script in the download directory on the server. If there is a script for download, it is downloaded and executed.

```
RET='emsg -n eemuserver \
    -p 1965 -w icecream -o query \
    -m "FILE dumbo.job" | \
    tee job_to_do.sh'
if [ "$RET" != "none" ];then
    chmod 700 job_to_do.sh
    job_to_do.sh
fi
```

In the above, the downloaded script is displayed by emsg on the standard output and subsequently saved in job_to_do.sh for later execution. If there is no script for download on the server, emsg returns string "none" and the body of the "if" statement is not executed. Notice that we do not need to log into the remote system to perform the task. For a site with many systems, the script or file distribution feature can provide great benefits for day-to-day system administration.

eEMU Action Scripts and Automation

Without action scripts, alarms can be viewed on the eEMU browser but there cannot be any automation. One of the major benefits of monitoring systems is that a corrective action or alarm escalation can take place automatically without human intervention. eEMU action scripts can be written in any programming or scripting language. The scripts are called in the background, whereby message attributes are passed to it as environment variables. Korn shell, Perl, Python, Tcl or Visual Basic are examples of excellent scripting languages that can be used with eEMU.

eEMU features three action scripts: input, output and delete. The input action script is called immediately on receipt of a new message. If the input script returns a value greater than 0, the message is discarded. The input script can be used for message replication or to supplement security and let in messages from specified hosts only.

The output action script is called for every message processed by eEMU except for delete messages. Delete messages invoke a separate script called "delete action script".

An exhaustive group of environment variables is provided for all the three action scripts, thus alarm actioning can be easily coded. There is no limit on what can be done in action scripts. `emsg` can also be used in action scripts to query the eEMU database, read a file or send a new message to eEMU. Care must be taken to avoid infinite message loops.

With a combination of resource IDs, message classes and message severity, call logging or paging is simple to achieve through action scripts. For example, to send a page if an alarm of severity 1 occurs, the following output action script can be used:

```
if [ $E_SEV -eq 1 -a $E_COUNT \
    -eq 1 -a $E_TYPE = "normal" ];then
    /usr/local/bin/page \
        $E_CLASS $E_HOST $E_MSG
fi
```

The first occurrence of a normal message (`E_TYPE`) with severity 1 (`E_SEV`) calls the `/usr/local/bin/page` script. The script can page the appropriate staff based on the message class (`E_CLASS`).

Using eEMU

While `syslog` and e-mail has been in use for some time and as such are powerful, proven messaging tools, they do not provide enough flexibility for dynamic monitoring in ever changing environment. With the messaging arsenal eEMU provides it is possible to build simple powerful agents not only for the system and hardware, but most importantly for applications. It is increasingly necessary to monitor various applications. This task is not easy with system agents provided with most other monitoring solutions. The eEMU messaging language is expressive enough to capture most typical scenarios in application monitoring. Such scenarios are dynamic masking out alarms, measuring application response times, catching overtime or hung business processes, identifying crashed scripts, synchronising batch job etc. Since eEMU agents do not, in most cases, have to store the resource status information, all that needs to be done is transform the monitoring case into the eEMU messaging language and if necessary, deploy the provided standard agents, e.g. the log agent, directory agent and process agent.

The eEMU server is a powerful implementation that has a minimum demand on operating system resources. E.g. the message database on disk does not typically grow beyond 500 kB. Action scripts are simple to implement. Action scripting is designed for easy integration with other products, such as paging software or third party messaging.

Status messages are equipped with a timestamp and written into a log file. There is one log file for

each day. The contents of the log file can be used for generating reports or ad-hoc alarm searches. Message attributes in the file are delimited with a vertical bar for handy processing by scripts or other reporting software.

The eEMU browser unifies the best capabilities of today's alarming concepts, namely color-coded system icons and event logs into one interface. As a result, the browser is extremely intuitive with minimum requirements on the operational staff. The latest version of the eEMU browser implements "business views" that allow to segment alarms into groups by function, geographical region, business importance etc.

There are a few passwords that enable access to specific message type groups. For example, helpdesk staff can view and annotate messages but cannot delete them. Each message type must be equipped with the correct password in order to be accepted by the server.

Since eEMU agents do not typically run as daemons, but rather are scheduled through a scheduling mechanism, such as `cron` on UNIX, eEMU represent a very safe monitoring solution for systems deployed in insecure environments such as firewalls. eEMUconcept provide an eEMU agent scheduler for Windows NT.

eEMU Integration

eEMU is a modular system that allows to build multi-tier hierarchies of monitoring agents and servers. The `emsg` executable forms the client side of the system. It can easily be integrated into any third party system. The integration can be done either at the `emsg` level or at the server level through action scripts.

eEMU has been successfully integrated with HP OpenView and Unicenter TNG. They are both SNMP based, so how was it done? Simply through action scripts. In case of standalone SNMP agents without an SNMP manager, there are a few approaches. An SNMP daemon can listen for SNMP traps and log them into a log file. The eEMU log agent is pointed at the log file and sends alarms for traps of our interest. If SNMP GETs need to be employed, a simple agent script, possibly written in perl, uses an `snmpget` utility to retrieve a resource status and subsequently sends a message to eEMU.

In the above, I described the case of alarms that are forwarded to eEMU for display on the eEMU browser and for triggering alarm actions. However, it is also possible to forward alarms from eEMU to a third party system. We can use the eEMU action scripts and command line interface as a vehicle for such forwarding.

Summary

In the network realm, SNMP is a successful, firmly established monitoring methodology. As a

result, there have been numerous attempts to apply SNMP to system monitoring. Our experience shows that such attempts are not successful. What administrators need is a powerful, simple command line interface to the monitoring server. Moreover, the interface needs to possess intelligence so that agent scripting requirements are reduced to the minimum. eEMU is a system that brings message type and time-to-live concepts that make the above possible. With eEMU, every system and network administrator is able to build monitoring and messaging infrastructure that forms the backbone of every enterprise management. There is no enterprise management without a powerful messaging foundation equipped with scripting interfaces at all levels. Some vendors like to say that enterprise management is using their products or their integration kits for other third party programs. This is not so. Enterprise management is based on open standard interoperability and easy integration hooks in and into existing systems. A lot of effort has been put into making vendors build their products to recognised standards. There has been more or less success doing that. However, command line interface to programs and script invocation on various events is still the best and most versatile API available.

Availability and Support

The eEMU software is available from eEMU-concept Pty Ltd, Australia. A free evaluation licence is available on request from the company's home web page (<http://www.eemuconcept.com>). The eEMU kit includes Unix and Windows NT system agents.

Author Information

Jarra Voleynik is a co-author of eEMU. He is currently working for eEMUconcept as a chief architect for the eEMU product. Jarra has a MSc degree in radioelectronics. He has been involved with computers, especially UNIX, for 12 years. He has worked, as a consultant, for major UNIX vendors, such as Digital Equipment Corporation, Compaq and Sun Microsystems. Recently, Jarra has architected one of the largest monitoring and call logging projects in Australia. Reach him electronically at jarra@eemuconcept.com.

Bibliography

- [1] Huntington-Lee, Terplan, & Gibson, *HP Open-View*, McGraw-Hill, 1996.
- [2] Knapik, M, & Johnson, J, *Developing Intelligent Agents for Distributed Systems*, McGraw-Hill, 1998.
- [3] Lendenmann, R, Nelson, J, Selby, J, & Patino Lara, C, *An Introduction to Tivoli's TME10*, Prentice-Hall, 1998.
- [4] Lirov, Y., *Mission Critical Systems Management*, Prentice-Hall, 1997.
- [5] Parsons T, Voleynik J, "Enterprise Event Management and Monitoring using eEMU and

Unicenter TNG (Open Source collaboration with Unicenter TNG," *AUUG 99 Open Source Conference Proceedings*, 1999.

- [6] Spuler, *Enterprise Application Management with Patrol*, Prentice-Hall, 1999.
- [7] Stevens W. Richard, *TCP/IP Illustrated, Volume 1*, Addison-Wesley, 1993.
- [8] Sturm, R., *Working with Unicenter TNG*, QUE, 1998.
- [9] Voleynik, J, Voleynik, A, "EMU - Event Management Utility," *Linux Journal*, November 1999.

Aberrant Behavior Detection in Time Series for Network Monitoring

Jake D. Brutlag – WebTV

ABSTRACT

The open-source software RRDtool and Cricket provide a solution to the problem of collecting, storing, and visualizing service network time series data for the real-time monitoring task. However, simultaneously monitoring all service network time series of interest is an impossible task even for the accomplished network technician. The solution is to integrate a mathematical model for automatic aberrant behavior detection in time series into the monitoring software. While there are many such models one might choose, the primary goal should be a model compatible with real-time monitoring. At WebTV, the solution was to integrate a model based on exponential smoothing and Holt-Winters forecasting into the Cricket/RRDtool architecture. While perhaps not optimal, this solution is flexible, efficient, and effective as a tool for automatic aberrant behavior detection.

Introduction

Real time management of a service network in frastructure at the IAP/ISP level is not a trivial task. First, there is the sheer quantity of data generated on a minute-by-minute basis. The WebTV production service infrastructure consists of tens of switches and routers, hundreds of host computers, and thousands of application daemon instances to support a subscriber base of over 1 million users. Second, there is great variety in the types of data collected. The WebTV production service monitors SNMP counters over network links, host statistics such as CPU load and I/O operations, and event logs for application daemons. Every variable monitored, whether byte traffic on a switch port, CPU load of a host machine, or requests handled by a cookie daemon, generates a time series. All of these time series reflect some part of the overall service network health.

The first challenge therefore, is to collect, store, and provide real-time access to this vast and diverse data. The open-source software RRDtool [6] and Cricket [1, 2] meet this first challenge. Using a web browser, a network technician can quickly navigate to and view a time series graph for a target and variable of interest.

The network technician is likely to be interested in aberrant behavior; that is, changes in the short-term behavior of a time series (on the order of minutes or hours) that are inconsistent with past history. Long-term trends (on the order of weeks or months) are not of interest from the service monitoring perspective because one expects a time series to evolve in a dynamic environment. Aberrant behavior may indicate a performance bottleneck, application component failure, or system downtime. In some cases, aberrant behavior is anticipated; other times it is not (see section "Defining Aberrant Behavior").

The second challenge of network monitoring is to automatically identify aberrant behavior in the midst of thousands of service network time series. Once such behavior is identified, an alert can be triggered to bring the technician's attention to the potential problem. Existing software tools provide some of this functionality, but these solutions usually rely on simple rules or thresholds (i.e., memory utilization should be below 80%). These rules and thresholds are sufficient for many applications, but they can't detect more subtle changes in behavior and they apply a static criteria to detect aberrant behavior rather than a dynamic one.

This paper describes a partial solution to this second challenge of network monitoring at the IAP/ISP level. Section "Description of the Model" discusses the aberrant behavior detection algorithm, with a focus on understanding the algorithm parameters. The bulk of the software implementation is in RRDtool, which is the focus of section "Enhancements to RRDtool." Section "Enhancements to Cricket" discusses details relevant for Cricket. The conclusion lists availability of the software.

Defining Aberrant Behavior

Suppose a statistical model exists that describes the behavior of a time series (or at least the characteristics of interest). With such a model, one can define aberrant behavior as behavior that does conform to the model (or is not well described by the model).

Of course, aberrant behavior with respect to a statistical model may or may not reflect a real event of interest for the technician. In the case that it does not, it is a false positive. Obviously, the ideal is to minimize the rate of false positives while identifying all events of real interest. However, this ideal can rarely be achieved. In most detection systems, there is a trade off between selectivity (avoiding false positives; also referred to as specificity and precision) and sensitivity

(ability to detect true positives; also referred to as recall). While it is important to remain cognizant of these issues, they become less important if one perceives a statistical model for aberrant behavior as a screening mechanism rather than a surrogate for the expert judgement of a network technician.

Note that this definition treats each time series independently of all others. No doubt there is much to be gained by leveraging the relationships between service network variables, but that is not addressed in this paper.

Description of the Model

Model Design Goals

Many service network variable time series exhibit the following regularities (characteristics) that should be accounted for by a model:

1. A trend over time (i.e., a gradual increase in application daemon requests over a two month period due to increased subscriber load).
2. A seasonal trend or cycle (i.e., every day bytes per second increases in the morning hours, peaks in the afternoon and declines late at night).
3. Seasonal variability. (i.e., application requests fluctuate wildly minute by minute during the peak hours of 4-8 pm, but at 1 am application requests hardly vary at all).
4. Gradual evolution of regularities (1) through (3) over time (i.e., the daily cycle gradual shifts as the number of evening daylight hours increases from December to June).

This list is by no means exhaustive; but it captures the most important characteristics.

In addition to modeling time series regularities, model design must consider the real-time monitoring context. Complicated statistical models are unlikely to be understood by network technicians and unlikely to be feasible computationally in a real-time context.

Overview of the Model

Aberrant behavior detection is decomposed into three pieces, each building on its predecessor:

- An algorithm for predicting the values of a time series one time step into the future.
- A measure of deviation between the predicted values and the observed values.
- A mechanism to decide if and when an observed value or sequence of observed values is 'too deviant' from the predicted value(s).

The proposed model is an extension of Holt-Winters Forecasting, which supports incremental model updating via exponential smoothing. The following sections discuss the model in some detail and require some mathematical notation. Let $y_1 \dots y_{t-1}, y_t, y_{t+1} \dots$ denote the sequence of values for the time series observed at some fixed temporal interval (recall RRDtool maps an irregular time series to a regular interval). Let m denote the period of the

seasonal trend (i.e., the number of observations per day).

Exponential Smoothing

Exponential smoothing [3] is a simple algorithm for predicting the next value in a time series given the current value and the current prediction. Let \hat{y}_{t+1} denote the predicted value for time $t+1$, then: $\hat{y}_{t+1} = \alpha y_t + (1 - \alpha)\hat{y}_t$

The prediction is actually a weighted average of all past observations in the time series. The premise of exponential smoothing is that the current value is most informative for prediction of the next value, and that the weight of older observation decays exponentially as the observation moves further into the past. It is an incremental algorithm because the next prediction is obtained by updating the current prediction with the current observed value.

α is the model parameter and $0 < \alpha < 1$. It determines the rate of decay $(1 - \alpha)$ and the weight the current value is given during the incremental update.

The Holt-Winters Forecasting Algorithm

Holt-Winters Forecasting [3] is a more sophisticated algorithm that builds upon exponential smoothing. Holt-Winters Forecasting rests on the premise that the observed time series can be decomposed into three components: a baseline, a linear trend, and a seasonal effect. The algorithm presumes each of these components evolves over time and this is accomplished by applying exponential smoothing to incrementally update the components.

The prediction is the sum of the three components:

$$\hat{y}_{t+1} = a_t + b_t + c_{t+1-m}$$

The update formulas for the three components, or coefficients a, b, c are:

- Baseline ("intercept"): $a_t = \alpha(y_t - c_{t-m}) + (1 - \alpha)(a_{t-1} + b_{t-1})$
- Linear Trend ("slope"): $b_t = \beta(a_t - a_{t-1}) + (1 - \beta)b_{t-1}$
- Seasonal Trend: $c_t = \gamma(y_t - a_t) + (1 - \gamma)c_{t-m}$

As in exponential smoothing, the updated coefficient is an average of the prediction and an estimate obtained solely from the observed value y_t , with fractions determined by a model parameter (α, β, γ). Recall m is the period of the seasonal cycle; so the seasonal coefficient at time t references the last computed coefficient for the same time point in the seasonal cycle.

The new estimate of the baseline is the observed value adjusted by the best available estimate of the seasonal coefficient (c_{t-m}). As the updated baseline needs to account for change due to the linear trend, the predicted slope is added to the baseline coefficient. The new estimate of the slope is simply the difference between the new and old baseline (as the time interval between observations is fixed, it is not relevant). The new estimate of the seasonal component is the

difference between the observed value and the corresponding baseline.

α , β , and γ are the adaptation parameters of the algorithm and $0 < \alpha, \beta, \gamma < 1$. Larger values mean the algorithm adapts faster and predictions reflect recent observations in the time series; smaller values means the algorithm adapts slower, placing more weight on the past history of the time series.

Note that the update formulas imply an implementation need only to store the current values of the slope and intercept, and a single period of seasonal coefficients, as these stored values are replaced at each iteration.

Holt-Winters Forecasting can also predict a time series further than a single time step in the future [3]. This multi-step prediction provides a mechanism to handle missing data.

Confidence Bands: Measuring Deviation

Confidence bands measure deviation for each time point in the seasonal cycle; this mechanism models seasonal variability. The measure of deviation is a weighted average absolute deviation, updated via exponential smoothing:

$$d_t = \gamma|y_t - \hat{y}_t| + (1 - \gamma)d_{t-m}$$

Here d_t is the predicted deviation at time step t . The update formula for d_t is similar to that of c_t . They even share the same adaption parameter, γ . The confidence band is simply the collection of intervals $(\hat{y}_t - \delta_- \cdot d_{t-m}, \hat{y}_t + \delta_+ \cdot d_{t-m})$ for each time point y_t in the series.

δ_+ and δ_- are scaling factors for the width of the confidence band. Often, a symmetric confidence band is desired and $\delta_+ = \delta_-$. In this case, denote the common parameter δ . Given some assumptions and statistical distribution theory, sensible values of δ are between 2 and 3 [7].

Aberrant Behavior Detection

A simple mechanism to detect an anomaly is to check if an observed value of the time series falls outside the confidence band. However, this mechanism often yields a high number of false positives. A more robust mechanism is to use a moving window of a fixed number of observations [7]. If the number of violations (observations that fall outside the confidence band) exceeds a specified threshold, then trigger an alert for aberrant behavior.

Formally, define a violation as an observation y_t that falls outside the interval:

$$(\hat{y}_t - \delta \cdot d_{t-m}, \hat{y}_t + \delta \cdot d_{t-m})$$

Define a failure as exceeding a specified number of threshold violations within a window of a specified numbers of observations (the window length).

Temporal Smoothing of Seasonal Cycle and Variation

As discussed thus far, each component of the seasonal coefficients vector is determined independently. It seems reasonable to assume that the seasonal

effect is a smooth function over the period, not a discontinuous series of points. A similar argument applies to the seasonal deviations. Note that exponential smoothing performs smoothing across seasonal cycles, but does not perform temporal smoothing within a seasonal cycle.

At a cost of adding some additional overhead to the implementation, the model performs temporal smoothing within a cycle for the seasonal coefficients and deviations. The smoother used is an equal-weight moving average, with a window of $0.05m$.

Choosing Model Parameters

The model parameters need to be set and tuned for the model to work well. There is no single optimal set of values, even restricted to data for a single variable. This is due to the interplay between multiple parameters in the model.

For example, consider two observations in sequence, y_t and y_{t+1} . The intercept (a), slope (b), and seasonal (c) coefficients all 'absorb' some part of the difference between y_t and y_{t+1} during the exponential smoothing update. It is safe to assume some of the difference is noise, so updates to the coefficients need not account for all of the difference between y_t and y_{t+1} . The values of α , β , and γ determine the relative share of the difference assigned to a changing baseline, a changing linear trend, and a changing seasonal coefficient

Here are some common sense guidelines for setting parameters:

- α : At least one of α , β , and γ should allow adaptation in a short time frame. As seasonal updates occur infrequently for each coefficient (once per cycle), and the goal of β is to capture a slowly changing linear trend, the most logical choice is α . Use exponential smoothing weights to make an educated choice for α . The sum of the most recent n weights is $1 - (1 - \alpha)^n$ and of course the sum of all weights is 1 (ignoring initialization, see section "Initialization"). These facts can be manipulated to choose α using the formula:

$$\alpha = 1 - \exp\left(\frac{\log(1 - \text{total weights as \%})}{\# \text{ of time points}}\right)$$

$\log()$ denotes the natural logarithm. For example, if one wants observations in the last 45 minutes to account for 95% of the weights, and observations occur at five minute intervals (nine time points), then the formula yields $\alpha = 0.28$. This formula can be rearranged using simple algebra to compute either the total weights as a percentage or the number of time points (elapsed time). For example, if $\alpha = 0.1$, then the most recent hour of observations at five minute intervals (12 time points) accounts for 75% of the baseline prediction.

- β : As the purpose of β is to capture a linear trend longer than one seasonal cycle, it is

logical to choose β such that one seasonal cycle does not account for a majority of the exponential smoothing weights. The formula discussed previously applies with β replacing α . For example, if the period of the cycle is one day at one observation every five minutes (288 per day), then setting $\beta = 0.0024$ will guarantee that observations within the last day account for less than 50% of the smoothing weights.

- γ : The seasonal adaptation parameter can also be selected using exponential smoothing weights using a variation of the previous formula. Note this single parameter controls both seasonal coefficient and deviation adaptation, on the assumption that seasonal trend and variability evolve together over time at roughly the same rate.
- δ : As noted in confidence bands section, the scaling factor of the confidence bands can be chosen by appealing to statistical distribution theory. Reasonable values fall in the interval [2, 3]. Choose 2 to detect more failures (which may just mean a higher rate of false positives).
- Window length and threshold: Given the goal of real-time monitoring, the window length should be at most on the order of an hour (i.e., for five minute intervals, choose a window length between 9 and 12). A higher threshold will make the model robust to false positives, but perhaps at the cost of missing true failures. These parameters are probably the most difficult to set a priori.

Initialization

The model requires initial values for the intercept, slope, seasonal coefficients, and deviations (seasonal variability). These could be set arbitrarily, computed from a long history of past data, or bootstrapped from data as it becomes available. The implementation in RRDtool is to bootstrap the algorithm from a cold start.

Initial values exert influence for some time. The analysis of exponential smoothing weights in the previous section assumes that influence of initial values has become negligible. For the intercept coefficient, the weight of the initial value in exponential smoothing for observation t is $(1 - \alpha)^{t-1}$. Similar formulas hold for β and γ . These formulas can be used to calculate the influence of initial values. For example, if the seasonal period is one day, 10 days have elapsed since initialization, and $\gamma = 0.1$, then the weight of the initial value in the predicted seasonal coefficient is 0.39. In contrast, the weight of the most recent observation (which in the long run is the most influential) is only 0.1.

Alternatives

While the model is designed to meet several goals, it is not optimal. The proposed algorithm lacks a formalism found in some other models. It is certainly true that there is no uniformly superior

forecasting algorithm for all time series, but consider the comments of researcher Richard Lawton [5]:

The Holt-Winters method is one of the best known forecasting techniques which allows the seasonal pattern to adapt over time... When compared with other methods the technique has been found to perform relatively well and it has the merit of being understood by users who lack a statistical background without sacrificing the ability to adapt to changing patterns in the data.

Enhancements to RRDtool

RRD is the acronym for Round Robin Database. RRD is a system to store and display time-series data [6]. It stores the data compactly, minimizes I/O operations for real-time updates, and presents useful graphs by processing the data at different temporal resolutions.

This section describes the implementation and usage of aberrant behavior detection in RRDtool. Some familiarity with the internals of the current release (1.0.x) of RRDtool is assumed, as this section makes reference to the pre-existing architecture.

Motivation

There are several reasons why support for aberrant behavior detection is integrated within RRDtool, as opposed to implemented as a standalone program. These include:

- Facilitates efficient real-time application of aberrant behavior detection. An external program would have fetch data from the RRD at the same frequency of update, while code within RRDtool is guaranteed to operate on this data already in-memory. Efficiency is a top priority for the service network at the IAP/ISP level, where RRDtool can be essential part of the monitoring system of hundreds of network interfaces and application services.
- Leverages ability of RRDtool to perform temporal interpolation (data updates at irregular intervals) and conversion of counters to rates.
- Leverages the graphing capabilities of RRDtool. Graphs relevant to aberrant behavior detection can be generated using the existing syntax of RRDtool.
- Leverages client software designed to run with RRDtool (i.e., Cricket).

Architecture

On disk, the round robin database (RRD) is organized into sequential sections, round robin archives (RRA). Within each RRA is a section for each of the data sources (variables) stored in this RRD. Each RRA is defined by a consolidation function which maps primary data points (PDP) to consolidated data points (CDP). At another level, an RRA is just an array of data values that is updated in sequence according to some function at some fixed time interval.

On its face, the aberrant behavior detection algorithm needs at least two arrays, one to store the forecast values corresponding to each primary data point, and a second to store the predicted deviation corresponding to each PDP. As implemented, the seasonal coefficients and deviations that are used to calculate the forecast and predicted deviations are stored in a second pair of RRAs. These arrays have length equal to the seasonal period and are updated once for each PDP. Failures are tracked by a fifth RRA, which determines violations and failures on each update.

The intercept and slope coefficients required for the forecast are updated for every primary data point and are unique for each data source. As only the most recent value of each is required (see “The Holt-Winters Forecasting Algorithm”), these parameters are stored in a temporary buffer in the header allocated for each RRA-data source combination in the RRD (the CDP buffer). This buffer is flushed back to disk on every call to RRD update.

The adaptation parameters are the same for all data sources within that RRA. They are stored in the RRA parameter buffer, which is read only during RRD update.

Therefore, implementation of the aberrant behavior algorithm adds five new ‘consolidation functions’ to RRDtool

HWPREDICT: an array of forecasts computed by the Holt-Winters algorithm, one for each PDP.

SEASONAL: an array of seasonal coefficients with length equal to the seasonal period. For each PDP, the seasonal coefficient that matches the index in the seasonal cycle is updated.

DEVPREDICT: an array of deviation predictions. Essentially, DEVPREDICT copies values from the DEVSEASONAL array to preserve a history; it does no processing of its own.

DEVSEASONAL: an array of seasonal deviations. For each PDP, the seasonal deviation that matches the index in the seasonal cycle is updated.

FAILURES: an array of boolean indicators, a 1 indicating a failure. The CDP buffer stores each value within the window. Each update removes the oldest value from this buffer and inserts the new observation. On each update, the number of violations is recomputed. The maximum window length enforced by this buffer is 28 time points.

Usage

This section illustrates how to use the aberrant behavior detection algorithm in RRDtool through an example. The monitoring target will be a router interface on a link between two data centers in the WebTV production service network. The variable will be the outgoing bandwidth rate (in Mbps). Bandwidth usage follows a daily cycle and SNMP is polled at five minute intervals.

Creating a RRD file

The first step is to create a RRD for this target with aberrant behavior detection enabled. In order to simplify the creation for the novice user, in addition to supporting explicit creation the HWPREDICT, SEASONAL, DEVPREDICT, DEVSEASONAL, and FAILURES RRAs, the RRDtool create command supports implicit creation of the other four when HWPREDICT is specified alone. To take advantage of this, use the following syntax:

```
RRR:HWPREDICT:<row count>:
    <alpha>:<beta>:<period>
```

Where:

row count is the number of forecasts to store before wrap-around; this number must be longer than the seasonal period. This value will also be the RRA row count for DEVPREDICT RRA.

alpha is the intercept adaptation parameter, which must fall between 0 and 1. The same value will be also be used for gamma.

beta is the slope adaptation parameter, again between 0 and 1.

period is the number of primary data points in the seasonal period. This value will be the RRA row count for the SEASONAL and DEVSEASONAL RRAs.

Using this implicit creation option creates the FAILURES RRA with a default window length of 9 and a default threshold value of 7. The default row count of the FAILURES RRA is one period.

Here is an example of the create command, using this syntax:

```
rrdtool create monitor.rrd -s 300 \
DS:ifOutOctets:COUNTER:1800:0:4294967295 \
RRA:AVERAGE:0.5:1:2016 \
RRA:HWPREDICT:1440:0.1:0.0035:288
```

After creation parameters can be changed using the tune command. RRDtool supports several new tune flags:

```
--alpha --beta --gamma
--window-length --failure-threshold
--deltapos --deltaneg
```

Each of these flags takes a single argument that corresponds to parameters discussed in section “Choosing Model Parameters.” The gamma flag will reset the adaptation parameter for both the SEASONAL and DEVSEASONAL RRAs (setting both to the same value). Both deltapos and deltaneg set the scale parameter for the upper and lower confidence band respectively, the default value for both is 2.

For example, suppose the technician is unhappy with the default window length and threshold for the FAILURES RRA implicitly created by the previous command. Issue the command:

```
rrdtool tune monitor.rrd \
--window-length 5 \
--failure-threshold 3
```

The remainder of the example uses the default window length of 9 and the default threshold of 7.

Other options of the create command, including syntax and details of the explicit creation of the new RRAs, are discussed in a detailed implementation document [4] and the RRDtool manual [6].

Detecting Aberrant Behavior

The aberrant behavior detection algorithm requires nothing unusual for the RRDtool update command; the collection mechanism (i.e., Cricket invoking SNMP) will run normally. Now suppose some time has passed and the network technician is monitoring outgoing bandwidth at the router interface. He can view a graph of daily activity, including confidence bands and any failures, with the command in Listing 1.

TICK is a new graphing option in RRDtool. For every non-zero value in the DEF or CDEF argument, it plots a tick mark. The length of the mark (line) is

specified by the third argument (after the color code) as a decimal percentage of the y-axis. 1.0 is 100% of the length of the y-axis, so the tick mark becomes a vertical line on the graph.

Figure 1 is an example of this daily graph generated on Wed, May 31, 2000 for the router target described previously. The thin lines are the confidence bands and the vertical bars represent failures (actually multiple failures in sequence – once the observed value strays outside the confidence bands it remains outside the bands for roughly a two hour period in both cases). The TICK graph option generates the bars from the FAILURES RRA.

The graph suggests that bandwidth on this outgoing link is increasing faster than expected by the model (past history). It is up to the network technician to decide if this represents aberrant behavior of interest. One approach the technician might take is to view the time series for this router interface over a longer time period.

```
rrdtool graph example.gif \
DEF:obs=monitor.rrd:ifOutOctets:AVERAGE \
DEF:pred=monitor.rrd:ifOutOctets:HWPREDICT \
DEF:dev=monitor.rrd:ifOutOctets:DEVPREDICT \
DEF:fail=monitor.rrd:ifOutOctets:FAILURES \
TICK:fail#ffffa0:1.0:"Failures Average bits out" \
CDEF:scaledobs=obs,8,* \
CDEF:upper=pred,dev,2,*,+ \
CDEF:lower=pred,dev,2,*,- \
CDEF:scaledupper=upper,8,* \
CDEF:scaledlower=lower,8,* \
LINE2:scaledobs#0000ff:"Average bits out" \
LINE1:scaledupper#ff0000:"Upper Bound Average bits out" \
LINE1:scaledlower#ff0000:"Lower Bound Average bits out"
```

Listing 1: Graph generation command.

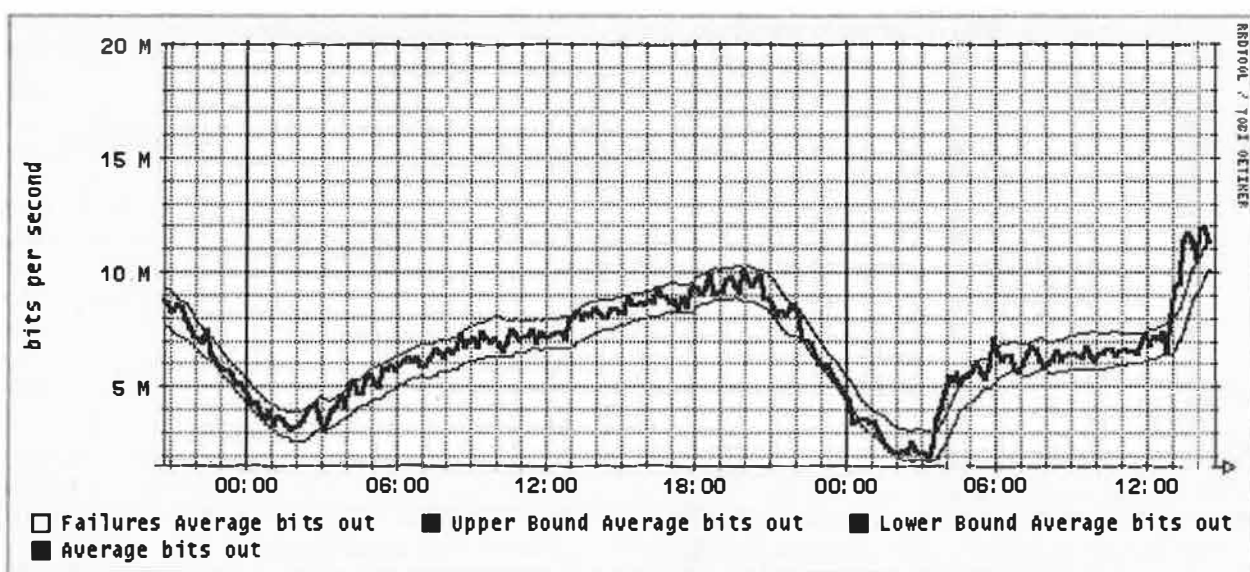


Figure 1: Observed bandwidth with Confidence Bounds for May 31

With hindsight, it is easy to demonstrate something unusual is going on and the aberrant behavior detection model is catching it in real time. Figure 2 is the time series for the week and half period from May 24, 2000 to June 2, 2000. It is clear that Wed, May 31, is unusual. Bandwidth increases in two steps: once in the early morning and again in the early afternoon. In this case, the dip to 0 in the early morning hour and the subsequent jump can be attributed to a scheduled downtime for the service network. Perhaps the remainder of bandwidth activity on Wed has the same cause, in which case aberrant behavior detected is a false positive in the eyes of the network technician.

Initialization

As alluded to in the previous initialization section, the implementation is designed to use bootstrap initialization. The intercept coefficient is initialized to the first observed value. The slope is initialized to 0, predicated on the assumption the linear trend over time is close to 0. If this is not the case, the time required for the Holt-Winters algorithm to gravitate away from 0 will depend on the seasonal adaptation parameter, gamma. During the first seasonal cycle of observed values, seasonal coefficients are initialized. During the second seasonal cycle of observed values, seasonal deviations are initialized. Unknown values during the first two seasonal cycles can complicate initialization. Basically, the implementation initializes any coefficients it can at the earliest opportunity; refer to the detailed implementation document [4].

Enhancements to Cricket

Cricket is a front-end to RRDtool [1, 2]. Cricket manages multiple time series via RRDtool in a target configuration hierarchy. The configuration hierarchy (or config tree) is a flexible approach to grouping targets with common time series variables, graphing characteristics, and other properties. The Cricket collector provides built-in and extensible mechanisms for

gathering data and feeding it to RRDtool. The collector manages SNMP calls and reads application event logs. The Cricket grapher generates time series graphs using the capabilities of RRDtool in real-time and serves them up as web pages. The graphs are organized via directory pages generated to match the config tree.

Monitoring

RRDtool has no mechanism for raising alerts, while Cricket does. Cricket provides several types of monitor-thresholds, which are defined in the config tree in a target dictionary section. Each monitor-threshold entry can contain multiple monitors. The basic functionality of a Cricket monitor-threshold is to fetch the most recent value from one of the RRAs of the target RRD file, check some criteria, and take some user-defined action if the criteria fails.

For efficiency and simplicity, Cricket 1.1 includes a new type of monitor-threshold specific for aberrant behavior detection. This monitor, failures, joins the existing Cricket monitors relation, value, and hunt. The general Cricket 1.1 monitor-threshold syntax permits a comma-delimited list of monitors. The syntax of each monitor is:

```
<data source>:<monitor type>:
<monitor args>:<action>:<action args>
```

The failures monitor does not take any arguments. For example, to send an SNMP trap whenever a failure is recorded for the ifOutOctets data source used in the example, the network technician adds the following entry to the appropriate target dictionary section:

```
my-monitor-threshold =
    "IfOutOctets:failures::SNMP"
```

Note that currently in Cricket 1.1, SNMP actions do not require any arguments, but the tag trap-address must be defined in the target dictionary. This may change in the future as Cricket 1.1 is still under development.

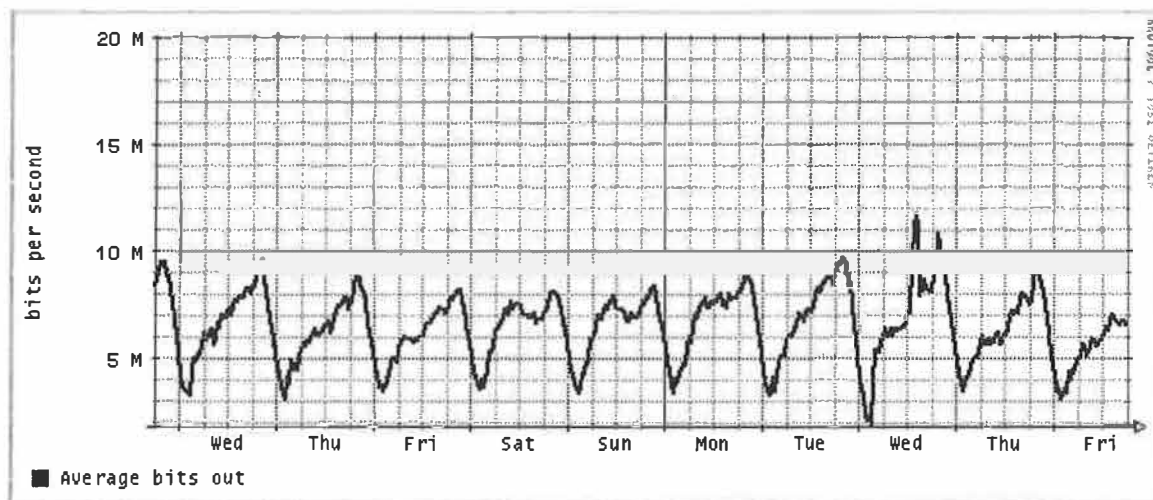


Figure 2: Router Interface bandwidth May 24-June 2

The failures monitor searches for a FAILURES RRA in the RRD file for the specified data source and if successful fetches the most recent value. If this value is 1, indicating a failure, it triggers the specified action. Through this mechanism, a network technician can easily be notified when the algorithm identifies something of interest.

Note that while the aberrant behavior detection RRAs, if created, apply to all data sources in the RRD file, the monitoring mechanism can be enabled for specific data sources or none at all.

HTML Navigation Links

Given the complexity of the RRDtool graph command, the Cricket 1.1 implementation provides a simple mechanism to view graphs relevant to aberrant behavior detection.

Cricket generates HTML navigation links to graphs using the new RRAs if it detects the string 'HoltWinters' in the name of the view. The view must consist of a single data source that is not multi-target. It verifies these restrictions before enabling the navigation links. There are three navigation links added: Confidence Bounds: displays the target data source using the Hourly Time Range with upper and lower confidence bands. The confidence band scaling factor (delta) cannot be obtained directly from the RRD file. Specify this factor with the confidence-band-scale tag in the graph dictionary. The default value (if the tag is omitted) is 2.

Failures: displays the target data source with prospective failures marked with vertical yellow lines (or yellow bars for failures in sequence) using the Hourly Time Range.

Confidence Bounds and Failures: the combination of the both of the preceding graphs.

Conclusion

There is a need to meet the second challenge of networking monitoring: automatic aberrant behavior detection. The model and software outlined here are a solid approach to the problem, working within the architecture of the existing open-source solutions RRDtool and Cricket. There is ample room for future work, especially in a solution that exploits not only the past history of a service network variable, but the relationships between service network variables.

Software Availability

The RRDtool implementation is available as a patch to the current release of RRDtool at <http://cricket.sourceforge.net/aberrant>. This web site also includes the more detailed reference document [4] on the implementation in RRDtool. The Cricket enhancements are part of Cricket 1.1, available at <http://cricket.sourceforge.net/>.

Author Information

Jake Brutlag is a statistician with the network operations group at Microsoft WebTV. He obtained

an MS degree in statistics from the University of Washington in 1999. He can be reached via email at jakeb@corp.webtv.net or U.S. mail at Microsoft WebTV; 1065 La Avenida; Mountain View, CA 94043.

References

- [1] Jeff R. Allen, *The Cricket reference guide*, <http://cricket.sourceforge.net/support/doc/reference.html>.
- [2] Jeff R. Allen, "Driving by the rear-view mirror: Managing a network with Cricket," *Proceedings of the 1st Conference on Network Administration*, 1999.
- [3] Peter J. Brockwell and Richard A. Davis, *Introduction to Time Series and Forecasting*, Springer, New York, 1996.
- [4] Jake D. Brutlag, *Notes on rrdtool implementation of aberrant behavior detection*, http://cricket.sourceforge.net/aberrant/rrd_hw.htm.
- [5] Richard Lawton, "How should additive Holt-Winters estimates be corrected?" *International Journal of Forecasting*, 14:393-403, 1998.
- [6] Tobi Oetiker, *The rrdtool manual*, <http://ee-staff.ethz.ch/oetiker/webtools/rrdtool/manual/index.html>.
- [7] Amy Ward, Peter Glynn, and Kathy Richardson, "Internet service performance failure detection," *Performance Evaluation Review*, 26:38-43, 1998.

PIKT: Problem Informant/Killer Tool

Robert Osterlund – University of Chicago

ABSTRACT

When faced with the many problems that arise in a complex of heterogeneous networked workstations, systems administrators often resort to coding scripts to monitor and problem-solve, scripts that they then schedule via cron. PIKT is a new and innovative approach to monitor scripting and managing system configurations. PIKT consists of an embedded scripting language with unique labor-saving features, a sophisticated script and system configuration file preprocessor, a scheduler, an installer, and other useful tools. More than just a systems monitor, PIKT is also a cross-categorical toolkit for configuring systems, organizing system security, formatting documents, assisting command-line work, and performing other common systems administration tasks.

The Problem, A Solution

Sysadmins have long wrestled with the task of writing generalized scripts to monitor systems and deal with recurring problem situations. As conventionally practiced, this approach has numerous disadvantages: it is hard to account for diversity across machines and operating systems; operations are fragile and error-prone; scripts for handling simple tasks are difficult to code, or are hardly worth the effort to maintain; scheduling and managing scripts are time-consuming and repetitive; setup is inflexible; activity and error logging is rudimentary or nonexistent; and the whole mass of scripts and configuration files is nearly impossible to keep track of or even comprehend.

PIKT attempts to solve some of the problems observed in more traditional methods of monitor scripting and managing system configurations. PIKT is an embedded scripting language and accompanying script interpreter. PIKT is also a sophisticated script and system configuration file preprocessor for use with the Pikt scripting language or any other scripting language of your choice. Finally, PIKT is a cross-platform, centrally run script scheduler (like cron), customizing installer (like rdist), command shell enhancement, and total script and configuration file management facility. PIKT's primary purpose is to monitor systems, report problems, and fix those problems whenever possible, but its flexibility lends itself to quite a few other uses as well.

Overview

In the usual PIKT configuration, you manage the monitored client ("slave") machines from a central ("master") control machine. On the central control machine, there are eight controlling config files: `systems.cfg`, `defines.cfg`, `macros.cfg`, `alerts.cfg`, `alarms.cfg`, `objects.cfg`, `programs.cfg`, `files.cfg`. They define your entire setup. You invoke the overseeing management utility, `piktc` (for "pikt control") to preprocess

those files, to install client target files, and to perform other management functions, such as stopping and restarting daemons.

Two daemons run on each client, `piktc_svc` and `piktd`. `piktc_svc` listens for and responds to `piktc` requests. `piktd` launches Pikt scripts at specified intervals.

On all clients, `piktd` wakes up every minute to check if one or more groups of scripts, also known as "alarms", are due to run that minute. Alarm scripts are grouped together as "alerts". Alerts run at specific intervals, e.g., hourly, once daily, once weekly, etc. At the appropriate time, `piktd` summons the `pikt` interpreter to run the Pikt scripts for that alarm group. You can also run Pikt scripts manually at the command line, but usually they are invoked by `piktd`.

`pikt` is the Pikt scripting language interpreter. Individual Pikt scripts usually monitor just one aspect of a system. You can monitor a single object, or collections of things listed in the object files, for example: system processes, disks, devices, lists, etc.

Each Pikt alarm script gets its input from processes, files, or log files. For log files, only new log entries since the last alarm run are considered.

A typical alarm consists of a sequence of logical tests. If the current input line satisfies one or more conditions, actions may or may not be triggered. Conditions might also refer to data in the previous input line, to data for this line remembered from the prior alarm run, even to data coming from outside the current alarm and `pikt` process.

Triggered actions might include generating a line of e-mail. At the end of the current `pikt` run, queued lines are e-mailed in a single problem report to one or more sysadmins. Queued lines might be printed or logged, whether to `syslog`, to this alert's log file, or to some other special log file. Or commands might be executed, for example to restart a detected dead system process, to chown a file, or perhaps to page the sysadmins.

For generating alarm script input, for taking action, also for serving as subroutines, you can employ auxiliary programs and scripts written in other, non-Pikt languages.

All external commands are logged for debugging and auditing purposes. If your alarm script makes reference to data from the prior alarm run, current data is stored in history files for looking up next time. And, very importantly, all errors are logged (including errors generated by invoked scripts written in other languages), giving you a complete audit trail when things go wrong.

You may also employ PIKT to manage system configuration files, such as `inetd.conf`, `syslog.conf`, `sudoers`, etc. It becomes much easier, for example, to enforce consistent access rights across your many systems.

The PIKT binaries are written using a combination of C, lex (flex), and yacc (bison). Most of the sample scripts are written in the Pikt script language, although several auxiliary Perl, expect, and shell scripts are also provided.

Configuration

Config Files

Every config file is a sequence of stanzas. A stanza consists of a stanza identifier, in the first column, then the stanza body, either on the same line or in multiple lines following. The stanza body must be indented, using spaces, tabs, or the `#indent` preprocessor directive.

Almost without exception, and aside from the above simple rules, PIKT is indifferent to script and config file layout. In other words, spacing and line breaks really don't matter, and you may lay out your config files in any style that pleases you.

PIKT comments are like those in C++, that is, `//` and `/* */`.

`systems.cfg` is where you specify host systems, host aliases, and host groups. Here is an example:

```

////////////////////////////////////
//
// PIKT systems configuration file
//
////////////////////////////////////

solaris
  hosts      moscow athens2
              berlin milan london
              paris paris4 paris5

linux
  hosts      murmansk firenze

***

milan
  aliases    bonn rome

***

```

```

mailserver
  members    moscow paris

***

```

`defines.cfg` specifies a set of “defines” – logical switches for including or excluding sections of the config files. Here are some example defines:

```

debug        FALSE
verbose      FALSE
paranoid     TRUE

```

`macros.cfg` specifies a collection of text substitutions. Macro definitions, but not macro names, may include embedded macros. Some examples:

```

behead(N)    =sed '1, (N)d'
offhours      ( #hour() >= 18 || \
                #hour() < 6 )

sysadmins     brahms|albeniz|liszt

```

In config files, a macro reference is preceded by an equal sign, for example: `=behead(1)`, `=offhours`, `=sysadmins`.

Macros may also include macro arguments. As with simple text-substitution macros, macros-with-arguments may reference other macros-with-arguments, so all manner of macro nesting is allowed.

In `alerts.cfg`, you schedule alarm scripts. Here is an example `alerts.cfg` stanza:

```

Urgent
  timing      0-45/15 * * * *
  drift       5
  priority    10
  mailcmd     "=mailx -s 'PIKT \
Alert on =host: \
Urgent' =pikturgent"

  lpcmd       "=lp =piktprinter"
  alarms      SysRebootUrgent
              FsMountsUrgent
              SwapChkUrgent
              ***

```

The timing parameters follow the usual cron conventions and then some. One not so usual timing spec is random timings (for example, `timing 20% * * * *`, which says to run the alert on average every five minutes). The random timing spec is especially useful in security situations, where you want some unpredictability in your monitoring schedules. Still another novel timing spec is “drift” – how many minutes an alert launch may randomly occur before or after a specified time – useful when you don't want alerts to “bunch up.”

As alarm scripts are run, their output is queued. At the end of the alert run (an alert is a set of alarms), the queued output may be sent as a single e-mail message to one or more sysadmins, or printed out, using the commands specified.

The `alarms.cfg` file is a series of Pikt scripts or alarm definitions. For a more detailed discussion of Pikt scripts, see the Scripting Language section below.

In `objects.cfg`, you specify system objects to be monitored. Object listings can also include data parameters. For example:

```
UserDirs
#if kiev2
    /pub/mus_disk_5
    /pub/mus_disk_6
#elif kiev0
    ***
#endif

***

SysProcs
    ***
    cron : /etc/init.d/cron start
    ***

***
```

The file `programs.cfg` contains support scripts written in other scripting languages, with each program in its own stanza.

In `files.cfg`, you can centrally manage system configuration files (such as `inetd.conf`, `motd`, and so on), and indeed any text file. `files.cfg` is much like `programs.cfg`, except that it can and should contain non-program files and/or programs external to the PIKT setup.

Partial Configurations

In a complete PIKT setup, you have all eight basic configuration files. You might, in addition, have `#include` file spinoffs from those basic eight.

It is possible to deploy PIKT in a partial configuration, with subsets of the eight basic config file types. `systems.cfg` is always required, but all the rest are optional.

Here are the most common PIKT setups:

- `piktc` as `rsh/ssh` replacement (no macros or defines): `systems.cfg` only
- `piktc` as `rsh/ssh` replacement (with macros and possibly defines): `systems.cfg`, `macros.cfg`; and optionally `defines.cfg`
- `piktc` as `rdist` replacement: `systems.cfg`, `files.cfg`; and optionally `programs.cfg`, `macros.cfg`, `defines.cfg`
- a centrally managed cron replacement: `systems.cfg`, `alerts.cfg`, `alarms.cfg`; and optionally `macros.cfg`, `defines.cfg`
- system/network monitor (but without system files management): all config files except `files.cfg`
- system/network monitor; `rsh/ssh`, `rdist`, cron replacements: all config files

So, you may utilize all that PIKT has to offer, or just pick and choose among its many functionalities.

Preprocessing

`piktc` & `piktc_svc`

PIKT is managed through the combined action of the interactive control program, `piktc` (on the central master machine only), and the `piktc_svc` service daemon (on all slave machines).

The `piktc` command options are shown in Appendix 1.

When specifying items, you include items with “+” and exclude with “-”. For example, “+A all” includes all alerts. “+A all -A EMERGENCY Info” includes all alerts except EMERGENCY and Info. Another way to achieve the same effect is with just “-A EMERGENCY Info” (leaving out the “+A all”, which is implicit).

This sample command checksums (using MD5) all files on all user systems except the Linux machines and any down systems:

```
# piktc -m5v ALL -H nonusersys
                        linux downsys
```

Preprocessing

You use `piktc` to preprocess source configuration (`*.cfg`) files on the master machine, and send the post-processing alert (`.alt`), object (`.obj`), program, and other files over the network to receiving `piktc_svc` daemons for installation on the slave systems. Preprocessing entails:

- stripping out meta-comments (comments of the form `//` or `/* */`)
- `#include`’ing auxiliary files (e.g., a list of Unix command macros)
- using `#if <os|host|hostgroup> #endif` preprocessor directives, filtering through lines pertaining only to the current client (e.g., `#if solaris`)
- using `#ifdef <define> #endif` preprocessor directives, for including/excluding portions of the text (e.g., `#ifdef debug`)
- making macro substitutions (e.g., substituting a Unix command path, with command options, appropriate to the current client)
- performing an across-the-board syntax check

Note that, in addition to the Pikt script and config files, it is possible to use meta-comments, `#include`’s, `#if`’s, `#ifdef`’s, and macros in managed system configuration files and scripts written in other languages (e.g., Perl [11], Python [6], AWK [2]). Note, too, that scripts may rewrite config `#include` files, raising interesting possibilities for maintaining dynamic system configuration files.

Preprocessor Directives

You can customize config files by means of the `#if`, `#elif`, `#else`, and `#endif` preprocessing directives. The format is

```
#if <machine class>
    <lines>
#elif <machine class>
```

```

    <lines>
#else
    <lines>
#endif

```

where <machine class> can be a series of host names, host aliases, or host groups, separated by the |, &, or ! set operators. | indicates set union, & set conjunction, and ! set negation. You can also use parentheses, (and), in the class specifications.

Akin to #if, a second class of preprocessor directives consists of: #ifdef, #ifndef, #elifdef, #elifndef, #elsedef, #endifdef, #define, and #undefine. The format is

```

#ifdef <define>
    <lines>
#elifdef <define>
    <lines>
#elsedef
    <lines>
#endifdef

```

where <define> is an identifier representing a type of logical switch that is either defined (true) or undefined (false).

Logical defines are set (to TRUE) or unset (to FALSE) in any of three ways: (a) in the file defines.cfg; (b) in any config file, except systems.cfg or defines.cfg, by means of the #define and #undefine directives; or (c) at the command line, by means of either the +D or -D switches.

Observe that you can set and unset defines on a per-machine basis in the defines.cfg file, for example

```

#if dbserver
paranoid      TRUE
#else
paranoid      FALSE
#endif

```

as well as nest #ifdef's within #if's, and vice-versa, throughout the config files.

A config file can incorporate one or more other files by means of the #include directive. Included files may themselves include other files, but only of the same basic configuration type (macro files include macro files, for example). Here is an example #include directive:

```
#include <security_alarms.cfg>
```

Includes are especially useful for compartmentalizing across different systems administrators (where each has his/her own sub-config file), and across functions (e.g., security alarms in one file, network alarms in another), and for including files contributed by outsiders. Includes are also good for quarantining information particular to different operating systems.

There are other preprocessing directives, but the ones described above are the most common.

Scripting Language

Script Outline

The general outline of a Pikt script is:

```

<script name>
  init
    status          active|
                   inactive
    level           emergency|
                   urgent|
                   critical|
    ...
    task            "<text>"
    input proc      "<process>"
                  file "<file>"
                  logfile "<logfile>"
    filter           "<process>"
    seps             "<char(s)>"
    dat              <var> <spec>
    ...
    keys            <var> [...]
  begin
    <statement>
    ...
  rule
    <statement>
    ...
  ...
end
  <statement>
  ...

```

Init Section

In the init section, you lay the basis for subsequent script actions.

The alarm is given one of eight severity levels, analogous to syslog's severity levels.

The primary alarm input is the output of a process, the full contents of a text file, or logfile updates (new info since the previous alarm run). If a process, it can be any system process (including multiple processes tied together by pipes) yielding text output. Pikt does not deal with binary input. Input may also be passed through an optional filter.

One or more dat statements map input data to variables. The dat statement takes one of three forms:

```

dat <var> x          [ordinal]
dat <var> x,y         [columnar]
dat "<regexp>"

```

For ordinal input, "seps" specifies a field separator (or separators) other than the default (whitespace).

Concluding the init section, the optional keys line lists variables used as database lookup keys when referring to history values (values stored from previous script runs).

Begin, End, and Rule Sections

Next come action statements, grouped into begin, end, and rule sections.

The heart of a Pikt script is the main processing loop: A line of input (from a proc, file, or logfile) is read in, then acted upon, the next line is read in and acted upon, and so on until the input is exhausted. Before input processing, you might have a begin section, to initialize some variables or take some other preliminary actions. You might also have an end section for input processing followup. (You can achieve additional data processing loops within a Pikt script using a combination of #fopen(), #popen(), #read(), and #fclose(), and/or #pclose().) In other words:

```

begin                                [optional]
    <statement>
    <statement>
    ***
[while there's input]                [optional]
    rule
        <statement>
        <statement>
        ***
[endwhile]
    end                                [optional]
        <statement>
        <statement>
        ***

```

The input processing loop consists of one or more rule sections. A rule section usually groups together program statements pertaining to a single attribute of the current input line. Strictly speaking, there is never a need to break up the set of input processing statements into separate rule sections, but doing so helps clarify program logic.

Data Types

Pikt has three basic data types: strings, numbers, and file handles (or proc handles).

Pikt supports both “associative” (string-indexed) and numeric (numerically-indexed) arrays. Array indices are computable (e.g., a concatenation, or the sum of two functions). Numerical arrays are (for now) limited to at most three dimensions. Note that, unlike with many other languages, Pikt array indices start at 1, not 0. (In Pikt, generally speaking, all indexing, in whatever context, begins with 1.)

Variables come in three different time forms. “\$” and “#” as variable prefixes refer to current values (strings and numbers respectively). “@” (e.g., @uid) signifies the value for this variable for the preceding input line. “%” (e.g., %usage) signifies the value for this variable during the previous script run.

So, in Pikt, there is no need to save input data values from one line to the next. Values from the previous input loop are stored automatically for you. The same is true with so-called “history variables.” Pikt stores values in a data file for recall the next time the alarm script runs. If a value is tied to a particular input data variable (specified in a dat statement) and a particular line of input, Pikt does a keyword lookup (specified in a keys statement) to find the appropriate data value.

Other Language Features

In general, every Pikt object serves a semantic purpose. Hence, and for example, parentheses are not required around an if condition or the arguments to a for statement. Nor are semicolons or end-of-lines required to signal the end of a program statement.

Pikt provides the usual operators, and a few not so usual. They mostly follow the Perl and AWK models.

Pikt offers a wide variety of built-in functions. An unusual feature of Pikt functions is that they are data-typed: their return value type is signified by either the “\$” prefix (for string; e.g., \$trim()) or “#” prefix (for number; e.g., #median()). Pikt does not currently support user-definable functions, although you can write pseudo functions using macros-with-arguments to achieve much the same effect.

Pikt comes with a panoply of flow control structures, most usual, and a few not so usual (e.g., ‘again’, for repeating the current rule; ‘leave’, for leaving the current rule). Every Pikt statement begins with a keyword (e.g., ‘set’, ‘if’, etc.). Statement blocks are indicated by a keyword-keyword combination, for example, if-endif, for-endfor.

Pikt uses AWK and GNU RX-style regular expressions.

Several Monitoring Examples

It should be emphasized that the examples following are not an intrinsic part of PIKT. They are solutions that you might implement, not that you are forced to adopt.

Case Study 1: IdleUserSession

IdleUserSession is a short Pikt script to kill abandoned user sessions. Listing 1 is the source version on the master control machine as it would appear in the alarms.cfg file.

We have decided that this needs to be run every other hour or so, so we group it with other “critical” alerts in the alerts.cfg file:

```

Critical
    timing          30 0-22/2 * * *
    drift            5
    #if moscow | munich
        priority     10
    #else
        priority     0
    #endif
    mailcmd          "=mailx -s 'PIKT \
                    Alert on \
                    =pikthostname: \
                    Critical' \
                    =piktcritical"

    alarms
        ***
        IdleUserSession
        ***

```

We would install this alarm, along with the other alarms in the Critical alerts group, with the command

```
# piktc -iv +A Critical -H downsays
processing madrid2...
installing file(s)...
Critical.alt installed
```

We have defined macro command paths in macros.cfg like so:

```
#if solaris
***
kill          /usr/bin/kill
***
nawk          /usr/bin/nawk
***
#endif
```

If the current client were defined as a solaris system in the PIKT systems.cfg file, the piktc preprocessor installs this script on the client (in the Critical.alt file) with the macros resolving to the appropriate solaris command paths, as in Listing 2, for example.

Note how macro substitutions have inserted the appropriate paths for the w, nawk, ps, and kill commands. If this were for one of the other supported operating systems, different paths would be inserted.

You no longer have to concern yourself with specifying the correct path for this or that command in your scripts, either by maintaining separate script versions or by inserting per-OS case statements into your

code. Simply define the path once and for all in the macros.cfg file, then use the =nawk macro (for example) ever after in all of your scripts (including scripts written in other languages, such as Perl, AWK, etc.). PIKT will automatically substitute the correct version for you.

Input data results from the command “=w”, i.e., “/usr/bin/w”. Here is a sample input line:

```
bach pts/4 29Jun98 3days 3:25 2 zsh
```

We pass this input along to nawk with the instructions: match lines showing idle time in days; transform, for example, “pts/4” into “ptsV4”; output just the first and second fields.

Pikt maps the nawk output “bach ptsV4”, setting \$user to the first field and \$tty to the second.

This alarm has but one rule: We exec a kill command to terminate the idle session in question. (The exec is automatically logged for auditing and debugging purposes.)

You could, if you want, add rules to kill root sessions only, or to kill after midnight and on weekends, or if certain other conditions are met. Instead of killing, you could send e-mail alerts to the system administrators, who could then decide if manual session kills are required.

Case Study 2: FileStatChk

One thing you would certainly want to monitor is the state of essential system files: Have they

```
IdleUserSession
init
    status active
    level critical
    task "Terminate idle user sessions."
    input proc "=w | =nawk '/[1-9]day/ {gsub("\\/", "\\V"); \
        print $1 " " $2}'"
    dat $user 1
    dat $tty 2
rule
    =execwait "=kill '=ps -ef | =nawk '/$user.+$tty/ {print \\$2}'"
```

Listing 1: IdleUserSession (source version).

```
IdleUserSession
init
    status active
    level critical
    task "Terminate idle user sessions."
    input proc "/usr/bin/w | /usr/bin/nawk '/[1-9]day/ \
        {gsub("\\/", "\\V"); print $1 " " $2}'"
    dat $user 1
    dat $tty 2
rule
    exec wait "/usr/bin/kill '/usr/bin/ps -ef | \
        /usr/bin/nawk '/$user.+$tty/ {print \\$2}'"
```

Listing 2: IdleUserSession (target version).

disappeared? Do they have the right ownerships and permissions?

We start by listing those files, together with their desired attributes, in objects.cfg (see Listing 3).

If we had adjusted the files list for the moscow system only, we would refresh the SysFiles objects set on that system with the command:

```
# piktc -iv +O SysFiles +H moscow
processing moscow...
installing file(s)...
SysFiles.obj installed
```

We could refresh all objects files on all active systems with the command

```
# piktc -iv +O all -H downsys
```

```
SysFiles
#if linux
    /etc/group          -rw-r--r--      644    root    root
    /etc/passwd         -rw-r--r--      644    root    root
    ***
#endif // linux
***
// local stuff
#if moscow
    /etc/mail/classalias -rw-r--r--      644    root    other
    ***
#endif
***
```

Listing 3: SysFiles.

```
FileStatChk

init
    status active
    level critical
    task "Detect critical file access deviations on system files."
    input file "=sysfiles_obj"
    dat $fil 1
    dat $prm 2
    dat $mod 3
    dat $own 4
    dat $grp 5
    keys $fil

rule
    if ! -e $fil
        output mail "$fil not found!"
    next
endif

rule
    do #split($list, $command("=lfd $fil"), " ")

rule
    if $list[1] ne $prm
        =execwait "=chmod $mod $fil"
        =outputmail "$fil permissions $list[1] are wrong" . \
            $if(#defined(%list[1])," (were %list[1]),",",") . \
            " changed to $prm"
    endif

[similar rules follow]
```

Listing 4: FileStatChk

It should be clear by now that the file /etc/mail/classalias would appear in moscow's Sys-Files.obj file and in no other system's.

Listing 4 is a script to enforce those file attributes.

For the first input line, "/etc/group" would be assigned to \$fil, "-rw-r--r--" to \$prm, "644" to \$mod, and so on.

In the first rule, if the file fails the existence test, that gets reported, and we move on to the next input line.

In the next rule, we take the output of the 'ls -l' command and #split() and assign the component parts to the \$list[] array.

In the third rule, if the actual file permissions, \$list[1], do not equal the desired permissions, \$prm, we fix and possibly report this.

The doexec define lets us control whether actions are exec'ed else a report of intent is e-mailed only. If this is a new PIKT installation, we might want to see what PIKT would do before committing PIKT to actually doing it. We could handle the conditionality this way:

```
#ifndef doexec
    exec wait "=chmod $mod $fil"
#elsedef
```

```
    output mail "=chmod $mod $fil"
#endifdef
```

But defining the following macro

```
execwait
#ifndef doexec
    exec wait
#elsedef
    output mail
#endifdef
```

in macros.cfg is more elegant, because now we can more succinctly write

```
=execwait "=chmod $mod $fil"
```

and either "exec wait" or "output mail" will be pre-processed in depending on how we defined doexec earlier.

In most circumstances, we simply want the file permissions fixed and don't need to be told about it. Sometimes, however, we want a full report of all that PIKT is doing. We control this by setting, in defines.cfg, the define verbose to be TRUE or FALSE. By defining the outputmail macro in macros.cfg as

```
outputmail
#ifndef verbose
    output mail
#elsedef
    output log "/dev/null"
#endifdef
```

```
rule
    output log "=swapchk_log" $inline()

end // only report if use is very high and increased by at
    // least 5% since last time (hence don't report when
    // swap use is high but declining)
    set #use = (#blksum-#fresum)/#blksum
    if ( #use >= 80% )
        && ( ( ! #defined(%use) )
            || ( %use < 80% )
            || ( #use - %use >= 5% )
        )
        output mail "swap utilization is $text(100*#use,0)%:=newline"
        output mail "swapfile          dev  swaplo blocks  free"
        for #i=1 #i<=#innum() #i+=1
            output mail $line[#i]
        endfor
        output mail =newline
        output mail $command("=dfk /tmp | =behead(1)")
        =dutop(10, /tmp)
        output mail "contents of /tmp:=newline"
        do #popen(LL, "=ll /tmp", "r")
            while #read(LL) > 0
                output mail $rdlin
            endwhile
            do #pclose(LL)
                output mail =newline
                =toptop(20)
            endif
        endif
```

Listing 5: SwapChk (fragment)

we can concisely write

```
=outputmail "$fil permissions
[...]"
```

If verbose is set to FALSE, the message is logged to /dev/null, that is, just thrown away.

Note the `$if(#defined(%list[1])," (were %list[1]),",")`. If we have run this script before, we have a record of the actual file permissions the last go-around in `%list[1]`. PIKT remembers this for us automatically. So if `#defined(%list[1])` is true, we report what they were, and in any case report what they have been changed to – but only if we have set verbose to TRUE.

Case Study 3: SwapChk

Another thing we monitor is if systems run out of swap space. For that purpose, we use the SwapChk script, a portion of which is shown in Listing 5.

The input for this script comes from input proc `"=swap -l | =behead(1)"`. The last rule above logs all input. This might come in handy some day if we need data to justify purchase of additional RAM.

At the end of all input, we compute `#use` as a percentage. If `#use` is equal or greater than 80%, or if `%use` is not defined (because this is the first alarm run, say), or if `%use` was less than 80% previously, or `#use` has gone up by at least 5% over the previous `%use`, we format a report and send it off as alert mail. Listing 6 is a sample report.

PIKT has assembled for us automatically all the diagnostic information we need to assess the situation. Moreover, after we have identified user freil as the memory hog, we can simply add some extra comments to the top of this alert e-mail and forward it along to freil – demonstrating one advantage of using e-mail as PIKT's primary notification mechanism.

We could also, at least under certain circumstances or on certain systems, augment swap space on the fly by adding the appropriate Pikt exec statements.

Case Study 4: ProcCountsChk

Recently, we have faced a crisis where a bug in the current version of our Web-based e-mail client has

```
PIKT ALERT
Thu Aug 17 21:20:14 2000
paris6
```

URGENT:

SwapChk

Report when swap use is high

swap utilization is 98%:

```
swapfile          dev  swaplo blocks  free
/dev/dsk/c0t0d0s1 32,1      16 1003184 24384
/pub/perf_disk_20/swap -      16 524272    0

swap              803568 757800 45768 95% /tmp
758376 /tmp/SAS_worka0000420D
8 /tmp/screens
240 /tmp/ups_data
```

contents of /tmp:

```
total 544
drwx----- 2 freil perf      629 Aug 17 21:18 SAS_work
drwxr-xr-x 2 root  other      69 Aug 16 06:15 screens
-rw-rw-r-- 1 root  sys    239160 Aug 16 11:12 ups_data
```

last pid: 17014; load averages: 0.20, 0.23, 0.23 21:20:21

54 processes: 46 sleeping, 3 zombie, 4 stopped, 1 on cpu

Memory: 128M real, 1576K free, 738M swap in use, 7984K swap free

```
PID USERNAME THR PRI NICE  SIZE  RES STATE  TIME  CPU CMD
16845 freil    1  35    0   12M 3336K sleep 4:27 9.28% r3
16909 freil    3  35    0 6432K 1464K sleep 1:37 5.21% sas
16969 root      1  33    0 4872K 2792K sleep 0:00 2.80% pikt
```

Listing 6: SwapChk (sample report).

```

ProcCountsChk
init
    status active
    level emergency
    task "Report unusually high counts of per-user procs."
    // note: a defunct process might show an empty comm field
    // below, so we pipe the ps output through the awk filter
    input proc "=ps -eo user,comm | =behead(1) | =awk 'NF==2' | \
        =sort | =uniq -c"
    dat #count 1
    dat $user 2
    dat $proc 3
begin // read in process and threshold data from objects file
    if #fopen(PROCCOUNTS, "=proccounts_obj", "r") != #err()
        while #read(PROCCOUNTS) > 0
            if #split($rdlin) == 5
                set #lgcnt[$1] = #val($2) // log thresholds
                set #alcnt[$1] = #val($3) // alert thresholds
                set #pgcnt[$1] = #val($4) // page thresholds
                set #klcnt[$1] = #val($5) // kill thresholds
            // else send an error message?
            fi
        endwhile
        do #fclose(PROCCOUNTS)
    else
        output mail "Can't open =proccounts_obj for reading!"
        quit
    fi
rule
    foreach #keys($pr, #lgcnt)
        if $proc =~ " $pr$ " // '=~', not 'eq', so that '\\*'
            // works as a default
            if #lgcnt[$pr] && #count >= #lgcnt[$pr]
                // for gathering diagnostic stats
                output log "=proccounts_log" $inline
            fi
            if #alcnt[$pr] && #count >= #alcnt[$pr]
                output mail $inline
                if $proc eq "imapd" // special case
                    =archive_mail_file($user, #true())
                fi
            fi
            if #pgcnt[$pr] && #count >= #pgcnt[$pr]
                exec wait "echo '=pikthostname: $inlin' | \
                    =mailx -s '=pikthostname: $inlin' \
                    =pagesysadmins"
                pause 5
            fi
            if #klcnt[$pr] && #count >= #klcnt[$pr]
                =kill_user_proc($proc, $user, #true())
            fi
        next // next input line
    fi
endforeach

```

Listing 7: ProcCountsChk

the imapd, under occasional and mysterious circumstances, spawning instances of itself every second or so. For a handful of users, we are seeing occasional “imapd storms” with per-user imapd counts reaching into the dozens, hundreds, and sometimes even thousands! At about the same time, but for different reasons, we began seeing “ypserv storms”. Not only do these storms risk losing user mail files, they also

imperil the entire system. Listing 7 is a Pikt script we have put into operation to deal with these sorts of problems.

The input proc statement yields input like

```

34 root /usr/lib/sendmail
404 chico imapd
1 zeppo imapd

```

In the begin section, we read data in from the ProcCounts.obj file (see Listing 8).

In the script's only rule, we check to see if the actual per-user process count exceeds the thresholds we set in the begin section, also if the threshold is non-zero.

Instead of `foreach #keys($pr, #lgcnt)`, we could have used for \$pr in #keys(#lgcnt). These accomplish the same purpose but with somewhat different syntax. Variety of expression and keyword synonyms are typical of Pikt. Did you notice the use of `if ... endif` in Case Studies 2 and 3 as opposed to `if ... fi` in the current case

```
ProcCounts
// 0 signifies take no action; 1 signifies always take action
//      proc          log          alert          page          kill
#   if moscow
#       imapd          10          100          1000          100
#   endif
#   if mailserver
#       sendmail       50          100          200          200
#   else
#       sendmail       5          10          20          40
#   endif
#   if nisserver
#       ypserv         2          3          3          0
#   endif
#       crack          1          1          1          1
#       sniffit        1          1          1          1
//      ...
// wild card should be last in ProcCounts list
//      \\\*          10          20          40          0
```

Listing 8: ProcCounts (objects.cfg fragment).

```
kill_user_proc(P, U, M)
// kill off all instances of a given process for a given user
// (P) is the process name (e.g., $proc, or "imapd")
// (U) is the user (e.g., $user, or "root")
// (M) is whether or not to output mail (e.g., #true())
set #killcount = 1 // initialize
while #killcount > 0
    set #killcount = 0
    do #popen(KILL, "=ps -eo pid,user,comm", "r")
    while #read(KILL) > 0
        if #split($readline) != 3
            cont
        fi
        if $2 eq (U)
            && $3 eq (P)
#ifdef debug
            output log "=proccounts_log" "$1, $2, $3"
            output log "=proccounts_log" "(P). (U), $text((M))"
#endifdef

            exec wait "=kill -9 $1"
            set #killcount += 1
        fi
    endwhile
    do #pclose(KILL)
endwhile
if (M)
    output mail "killed all (U) (P) processes"
fi
```

Listing 9: kill_user_proc()

study? Another example: `elif`, `elsif`, `elseif` are synonymous, and all achieve identical effect.

If the `#lgcnt[]` threshold is non-zero and if the process count exceeds the `#lgcnt[]` threshold, we log some diagnostic statistics for post-mortem analysis. If the process count exceeds `#alcnt[]`, we send alert mail reporting that fact. In the case of `imapd` only, we also

backup the user's mail file by means of the `=archive_mail_file()` macro (not shown).

If `#count` exceeds `#pgcnt[]`, we send a short alert message to `=pagesysadmins`, a macro that resolves to the `sysadmins`' pager numbers.

Finally, if `#count` exceeds `#klcnt[]`, we kill off the user processes by means of the `=kill_user_proc()` macro (see Listing 9).

```
UserActivity

    init
        status active
        level critical
        task "Report and/or log suspicious after-hours activity."
        input proc "=w -hs"
        =wdata

    begin
        exec wait "=touch =useractivity_log" // forced update
    #ifdef worried // or paranoid
        if #false() // never quit this alarm if we're
                    // worried (or paranoid); monitor
                    // activity at all hours
    #elsedef
        if #hour() >= 8 // 8 AM to midnight only
    #endifdef
        quit // don't monitor, move on to next alarm
    endif

    rule
    #ifdef worried // or paranoid
        if #true() // all users
    #elsedef
    # if nonusersys
        if #true() // all users, on admin systems
    # else
        if $user eq "root" // root only, on user systems
    # endif
    #endifdef
        && ( #length($idle) == 0
            || $idle =~ "[0-9]+$" // idle time in minutes,
                                // not hours or days
        )
        // escalate notification at higher levels of security
        output log "=useractivity_log" $inline
    #ifdef cautious // or worried or paranoid
        output syslog $inline
        output mail $inline
    #endifdef
    #ifdef worried // or paranoid
        output print $inline
    #endifdef
    #ifdef paranoid
        exec wait "echo 'pikthostname: $inlin' | =mailx -s \
                    '=pikthostname: $inlin' =pagesysadmins"
    #endifdef
    endif
```

Listing 10: UserActivity.

Here is a sample alert message:

```
PIKT ALERT
Tue Apr 25 00:17:02 2000
moscow
```

URGENT:

```
ProcCountsChk
Report unusually high counts \
of per-user procs.

404 chiko imapd
saved user mail file as
/var/mail/arc/chiko.956639822
killed all chiko imapd \
processes
```

(We still don't have an understanding of these problems, much less fixes, but at least we are not losing any more user e-mail, and our mail server is coping.)

Before leaving ProcCountsChk, note that by defining all count thresholds to 1 across the board, we can guard against users running "dangerous" or "forbidden" programs such as Crack or Sniffit.

Case Study 5: UserActivity

You can use PIKT's define feature to achieve precision control over your security setup. Consider these security settings in defines.cfg:

```
attentive TRUE // lowest level
// of security
```

```
cautious
#if misscritsys | cssys
TRUE
#else
FALSE
#endif
worried
#if misscritsys
TRUE
#else
FALSE
#endif
paranoid FALSE // highest level
// of security
```

Listing 10 shows how you might use them in an alarm to monitor suspicious, after-hours user activity (some per-OS customizations were omitted for brevity).

We can also apply these defines to the UserActivity.log file produced by the UserActivity alarm. Here is a sample log entry:

```
Aug 3 01:36:01 CRIT: root p0 1 -csh
```

Listing 11 shows the log monitoring script.

As security conditions change, we can generate more or fewer log entries by changing our security

```
UserActivityLogChk
init
status active
level critical
task "Report all new security incidents in UserActivity log."
input logfile "useractivity_log"
#ifndef cautious // or worried or paranoid
begin
quit
#endif
#ifdef cautious // or worried or paranoid
rule
output syslog $inline
#endif
#ifdef worried // or paranoid
rule
output mail $inline
#elsedef // only cautious
rule
if $inline =~ "root"
output mail $inline
fi
#endif
#ifdef paranoid
rule
output print $inline
// page also?
#endif
```

Listing 11: UserActivityLogChk.

defines (from TRUE to FALSE, or vice-versa) for different systems, then using `piktc` to reinstall the modified scripts on those systems. This gives us pinpoint control over our security setup.

Other Uses

Those are just a very few of the things you can use PIKT to monitor. We use it for all kinds of systems administration tasks, including: clearing out /tmp files; reporting system crashes; monitoring changes in critical system files, directories, and devices; detecting passwd and shadow file anomalies; running a mail quota system; reporting “orphaned” accounts and home directories; detecting bad e-mail list addresses; clearing out user Web browser caches; removing core files; rotating and retiring system log files; reporting full file systems; reporting runaway processes; reviving vital system processes; reviewing security log files – the list goes on and on.

Working with Other Scripting Languages

If you prefer to use a different scripting language, that is no problem. Here is a short Pikt wrapper script around a much longer, and very complicated Perl script, `=mailchk (/pikt/lib/programs/mailchk.pl)`:

```
MailChk
  init
    status active
    level warning
    task "Check for mail \
        errors, such as \
        forwarding loops."
    input proc "=mailchk 2>&1"
  rule
    output mail $inline
```

`mailchk.pl` yields output which the Pikt MailChk script captures in a PIKT e-mail alert. If you wish, you could let your Perl script handle all the reporting, but still have PIKT deal with scheduling and logging, using the minimalist Pikt script:

```
MailChk
  begin
    exec "=mailchk 2>&1"
```

We have a suite of over two dozen account management programs, almost all of them written in Perl, that we maintain within our `programs.cfg` file. We don't use Pikt or `piktd` at all to run these. Rather, we

```
#if usersys
  ftp      stream tcp      nowait  root    /usr/tcpd/tcpd  in.ftp
  telnet    stream tcp      nowait  root    /usr/tcpd/tcpd  in.telnetd
  ...
#else
  #ftp      stream tcp      nowait  root    /usr/tcpd/tcpd  in.ftp
  #telnet    stream tcp      nowait  root    /usr/tcpd/tcpd  in.telnetd
  ...
#endif
```

Listing 12: `files.cfg` (fragment).

use PIKT to manage the per-OS and per-machine differences, to install, and to monitor script integrity.

Config Files Installation and Management

Listing 12 is a portion of our `files.cfg`, the section configuring `inetd.conf`.

Turning services on and off is as easy as editing the central `files.cfg`, then reinstalling `inetd.conf` with the appropriate `piktc` command.

Recently, a CERT advisory was broadcast advising against running the `rpc.ttdbserverd` service with root privileges. For the `rpc.ttdbserverd` line, we substituted “daemon” for “root” (the line was already commented out anyway), then updated `inetd.conf` and reconfigured `inetd` on all Solaris systems with:

```
# piktc -iv +F inetd.conf \
    +H solaris -H downsys

# piktc -xv +S SigHupInetd \
    +H solaris -H downsys
```

where `SigHupInetd` is a Pikt script written expressly for that purpose.

Another problem we have faced is keeping up-to-date our `sudoers` file – especially the list of part-time Computer Assistants. We do it in `files.cfg` by means of an include file:

```
User_Alias      PARTTIMERS=\
#include <sudo_parttimers_files.cfg>
```

where the `sudo_parttimers_files.cfg` file might be:

```
larry,moe,curly,sporty,\
ginger,baby,pose,scary,\
john,paul,george,ringo
```

We have a separate script that rewrites the `sudo_parttimers_files.cfg` file nightly based on an authoritative and up-to-date GNU Mailman list. The result: a dynamic `sudoers` config file!

Remote Command Execution

You can use `piktc` for remote program execution as an alternative to `rsh` or `ssh`. The command

```
# piktc -Xv +C "<command(s)>" \
    +H <systems>
```

executes the given `command(s)` on the specified systems.

You can insert PIKT macros within +C command strings. See Listing 13, for example.

Note that kiev0 is a SunOS system. We want 'df -k' to run on the Solaris systems and just plain 'df' to run on the SunOS systems. The macro =dfk resolves to the desired path and command option.

System Lists

With perhaps the simplest but still useful PIKT setup imaginable – the piktc binary and a systems.cfg file – you can maintain custom system lists, whether for referencing within other programs, as in this Perl statement

```
@hpsys = 'piktc -L +H hpux \
          -H downsys';
```

or for command-line work, as in a command loop we use to upgrade our Solaris PIKT binaries (see Listing 14).

The uses of PIKT really are limited only by your imagination!

Security

The current PIKT security model is fairly trusting. There are things you can do now to tighten security, while other things – Kerberos-style client-server authentication and data encryption, for example – are under study and planned for implementation in the near future.

In PIKT.conf, which functions roughly like a .rhosts or .shosts file, you set various access parameters (e.g., uid, gid, master, domain, master address, etc.) and service rights (e.g., all_services, kill_service, install_service, etc.). By careful and judicious use of these settings, and in combination with other measures (firewalling, running piktc_svc only when necessary, etc.), you can achieve a level of security sufficient in many situations.

When you issue a piktc command, the slave (remote) host and requested service are registered with the master (local) piktc_svc. Upon receiving the

service request, the slave piktc_svc checks its local access authorizations in PIKT.conf. If the service request is authorized, the slave piktc_svc then does a “callback” – a second, independent TCP connection – to the master piktc_svc, seeking to verify the request. If it verifies – i.e., both slave host and service request match – only then does the slave piktc_svc perform the requested service. It then returns the outcome of the service request to the requesting piktc. Finally, the slave host and requested service are deregistered on the master piktc_svc. Please see Figure 1.

piktc and piktc_svc do complete service request logging, and piktc_svc marks denied requests as “ERROR”. It is not difficult to set up log monitoring scripts (and scripts to monitor the monitoring scripts, etc.) to spot attempted security break-ins. Remember that you can log to syslog and special logs, and dump to a printer, besides sending out e-mail alerts. Scripts can also call pagers in security emergencies. You can even have a monitoring script kill the local piktc_svc under suspicious circumstances.

You can use PIKT for security in at least the following six ways:

1. As just a centrally managed scheduler. Have piktd invoke your preferred security tools according to the schedules in piktd.conf.
2. The above, plus have PIKT manage other security tools' config files (the inevitable per-machine and per-OS customizations).
3. All of the above, plus use PIKT #ifdef's to activate and deactivate different security tools as changing conditions warrant.
4. All of the above, plus use PIKT to handle your security log file analysis and incident reporting.
5. All of the above, plus employ PIKT alarms and data objects as supplements to the standard tools. (For example, have PIKT do things that COPS or Tiger don't do.)
6. Use PIKT exclusively.

```
# piktc -x +C "hostname; =dfk /tmp" +H mus
kiev0
Filesystem      kbytes    used    avail capacity  Mounted on
/dev/sd0e       993006      17   893689      0%      /tmp
kiev
Filesystem      kbytes    used    avail capacity  Mounted on
swap           769096     296   768800      1%      /tmp
...
```

Listing 13: piktc Command Example

```
# for sys in 'piktc -L +H solaris -H piktddevsys no_usr_local downsys'
> do
> echo $sys
> ssh $sys "/pikt/lib/programs/svcstart.pl -k; \
cp /pikt/bin/pikt* /pikt/bin/bak; \
cp /usr/local/pikt/bin/solaris/pikt* /pikt/bin; \
/pikt/lib/programs/svcstart.pl -r"
> done
```

Listing 14: System Lists

PIKT is especially adept at points one, two, and three above.

As for point four, there are good solutions out there already for analyzing your security log files, but PIKT might be superior due to its more powerful and flexible built-in scripting language.

As for point five, extending one tool requires you to learn Perl, another to get intimate with the Bourne shell, while five others require you to learn five different cryptic and proprietary command languages. Learn those languages and modify those tools on their own terms where that makes the most sense, but when sensible resort to PIKT.

As for using PIKT exclusively, in spite of its great virtue of involving just one system and command language to learn and use, I don't propose that PIKT do everything! For many purposes, there are some really excellent alternatives available. I therefore suggest that the optimum solution lies somewhere around points three, four or five.

Security should be systematic, flexible, and easy to manage. These are areas where PIKT excels. At the very least, PIKT is a general framework on which to build your security efforts.

Future Plans

PIKT could use a GUI, not so much to overlay pikt management as to handle incoming alert messages. This could take form as, for example, a Tk/Tcl or Java front-end acting on syslog messages sent to the console machine.

Although there are methods to program against endlessly repeating nuisance messages (so-called "nagmail"), PIKT would benefit from more automated ways to do this. Message routing could also be improved.

PIKT has a steep learning curve, and setup can be daunting. A project is underway to put together a PIKT "standard library" of ready-to-run defines, macros, alerts, scripts, programs, and objects sets. We need to improve the user's "out-of-the-box" experience.

The weakest link in PIKT's chain is the script interpreter, pikt. Although it gets the job done, it is not

fast or feature-complete. Pikt is not a stand-alone, all-purpose scripting language, and you cannot effectively call Pikt scripts from programs written in other scripting languages. Pikt was designed to aid in systems administration and to run "fast enough" in a small memory space within the broader PIKT system. Plans are to rewrite Pikt's underlying engine, perhaps using GNU Guile or embedded Perl.

PIKT requires a thorough security audit, and its client-server communications need to be made iron-clad secure. Adding encryption and Kerberos authentication is under consideration. We also plan to implement a comprehensive security package of PIKT defines, macros, scripts, and objects sets to work alone or in concert with other security products.

PIKT has an Introduction and comprehensive Reference Manual but lacks a Getting Started guide, also an Operations Manual.

Other Solutions

PIKT is often compared to Cfengine [3]. In the words of its author, Mark Burgess,

Cfengine ... is a very high level language for building expert systems which administrate and configure large computer networks. Cfengine uses the idea of classes and a primitive form of intelligence to define and automate the configuration of large systems in the most economical way possible.

In Cfengine, you create a configuration file (or files) describing the ideal setup for all of your hosts. When run, the cfengine program will check the actual machine configurations against the ideal and, if desired, fix any deviations.

Cfengine and PIKT address generally the same problem but in significantly different ways. Cfengine is a high-level, declarative or descriptive language (a single statement might set permissions on hundreds of files, for example), while Pikt is a low-level, procedural language. Cfengine tends to provide specific solutions to specific problems, while PIKT tends to be more general. Cfengine's specificity (it has built-in support for configuring network interfaces, for example) would be out of place in base PIKT. (With PIKT,

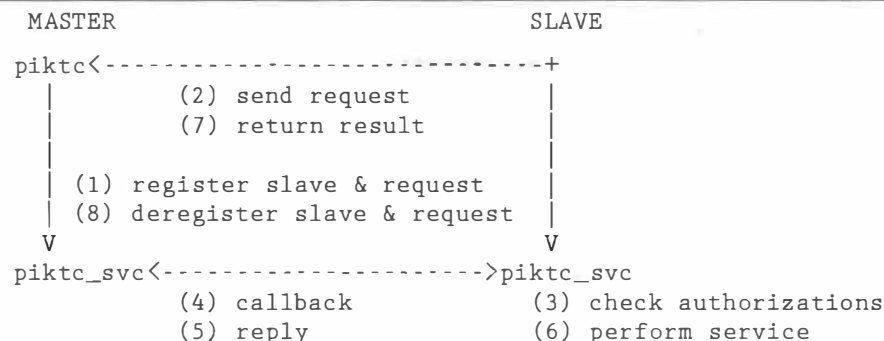


Figure 1: callback.

you would write a script to configure the network interface calling the usual UNIX networking commands.)

While Cfengine achieves per-OS and per-machine customization by means of classes, for example,

```
FTPserver.Sunday.Hr00::
    /local/iu/xferlog rotate=3
```

which means to run xferlog at midnight Sunday if this system is a FTPserver, PIKT would achieve a similar effect as follows (in the alerts.cfg file):

```
Notice
    timing      0 0 * * 0
    ***
    alarms
    ***
#if ftpserver    LogFileChkNotice
#endif
    ***
```

Cfengine has its own unique keywords, syntax, macro and variable forms, etc. Although Pikt has some unique elements, much of it should be familiar to any Perl or C programmer, especially the idea of preprocessing. If PIKT has a steep learning curve, Cfengine's is equally steep, if not steeper.

Cfengine tries to anticipate many of your needs, but when you veer off the beaten path, Cfengine is not quite so helpful. In many situations, you will still need to write your own scripts. With PIKT, you script everything. This makes PIKT inherently more flexible and applicable to a broader class of applications – not just fixing broken system configurations and executing routine tasks, but also reacting to errant dynamic processes.

Cfengine is quite good at what it is designed to do. It would be especially useful (and superior to PIKT) for configuring a new system or restoring a system after a crash or cracker break-in. One really nice Cfengine feature is that ordinary users can invoke it, attempting to fix a broken configuration if the system administrator is unavailable. (PIKT is typically just for root use.)

In work first presented at the LISA 1999 Conference [4], Alva Couch and Michael Gilfix have

... created a system administration library that allows one to perform system administration tasks in Prolog. This is much more powerful and flexible than using other current tools, and has the advantage that the resulting Prolog programs are much closer to describing actual policies than CFEngine configuration files or PIKT scripts.

Perhaps because their comments were based on earlier, less mature versions of PIKT, I feel that they underestimate the power, flexibility, and expressiveness of Pikt scripts, especially the fully documented,

macro-enhanced versions found in the central configuration files (as opposed to the preprocessed, uncommented versions installed on the slave systems).

Their Prolog-based approach to systems administration is intriguing and potentially far-reaching, but it suffers from one significant problem: Unless one attains proficiency with Prolog (not a widely used language, to say the least), their system is a “black box,” closed to the do-it-yourselfer who demands complete control over, or at least complete understanding of, the system. In any case, at this time, source code is not yet available for public distribution, so it is hard to evaluate their approach effectively.

There are other systems monitoring packages out there, including: Big Brother [7], and its clone Big Sister [1]; Mon [9]; NetSaint [5]; and still others. These tend to focus more on performance statistics and problem reporting, less on systems configuration and problem solving. To their credit, they rely on standard scripting languages, but they don't deal specifically or as extensively with the problem of per-machine and per-OS customization like PIKT, Cfengine, and the Prolog-based library do.

I have no experience using any of the high-octane, very expensive commercial packages (like Tivoli [8] or CA Unicenter TNG [10]) and can't venture any comparisons or opinions about them.

Parting Thoughts

The heart and soul of PIKT is its preprocessor, pikt, and all the special scripting and file management facilities it provides: per-machine and per-OS #if directives; the #ifdef family of logical switches; #include files; macros; pinpointed file installation; central scheduling; and so on. PIKT moves scripting toward the kind of full-featured development environment that users of “more serious” languages have long enjoyed.

The Pikt scripting language offers some unique features, or features better tailored to the job of day-to-day systems administration: automatic previous-line (@foo) and prior-run (%foo) value references; a clean, uncluttered syntax; free-form, flexible layout; keyword synonyms; automatic logging of everything of consequence; a cautious approach to script execution (no assumed variable defaults; serious errors trigger automatic script shutdown); a built-in input loop (much like AWK's); many standard input and output options.

On the other hand, people have a right to question whether the world needs Yet Another Scripting Language. Also, scripting language preference is often a highly personal, even emotional matter. Pikt, the scripting language, is just the first among equals in PIKT, the sysadmin toolkit. Use of other languages within the PIKT system is encouraged, and embedding other scripting languages within PIKT is actively being considered.

But the point bears repeating: the piktc preprocessor and control program is at the center of PIKT. It is what sets PIKT apart from other scripting languages and other system monitors. PIKT is more than just another systems monitor and Yet Another Scripting Language. When confronted with its multi-functionality, one Web administrator didn't quite know where to list it, saying that he might have to invent a whole new category for PIKT. If ever a tool were more than the sum of its parts, PIKT is that tool. The PIKT combination is a very powerful, wide-ranging, and ambitious toolkit indeed.

Availability

PIKT's homepage is: <http://pikt.uchicago.edu/pikt>, where you will find not only the distribution package but also complete on-line documentation, sample configuration files, a comprehensive test suite (with over 600 validation tests), and other useful items. PIKT is also available for download at a number of ftp sites. `pikt-users` and `pikt-workers` mailing lists have been formed. Operating systems now supported include GNU/Linux, Solaris, SunOS, FreeBSD, OpenBSD, AIX, HP-UX, and IRIX.

Acknowledgments

I need to thank the following persons for their helpful criticisms, suggestions, bug reports, and in some cases code: Bardur Arantsson, Michel Blanc, Jim Botts, Leon Breedt, Chris Halverson, Magdalena Hewryk, Rich Hoffer, Kelsang Wangden, James Low, David Masterson, Miguel Armas del Rio, Roland Roberts, Raul Alexis Betancort Santana, Mike Scheidler, Joe Siegrist, and especially Harlan Stenn, who implemented the PIKT `autoconf/automake` and who has helped out in other innumerable ways. I owe a huge debt of gratitude to the authors and maintainers of `gcc`, `gdb`, and `make`, as well as GNU `flex` (`lex`) and `bison` (`yacc`), upon which PIKT relies quite heavily. I am also very grateful to Will Partain and Kelsang Wangden for providing thoughtful and incisive suggestions for improving this paper.

Author Information

In a former life, Robert Osterlund earned a couple of economics degrees from the University of Chicago and worked as an economist and college teacher while serving as a U.S. Peace Corps Volunteer in the Philippines. After making a mid-life career switch to computing, he took computing courses for a while at the University of Illinois at Chicago, then returned to the Philippines to organize and head the computer department at a small college there. He was employed for five years as Senior Programmer Analyst at the University of Chicago's Social Sciences and Public Policy Computing Center, and has worked as Unix Systems Manager at the University's Graduate School of Business since 1995. You can mail him at: Robert Osterlund, Graduate School of Business,

University of Chicago, 1101 E. 58th Street, Walker 309, Chicago, Illinois 60637, USA. Or send e-mail to: robert.osterlund@gsb.uchicago.edu.

References

- [1] Thomas Aebly, Big Sister, <http://bigsister.graeff.com/>.
- [2] Alfred V. Aho, Brian W. Kernighan, Peter J. Weinberger, The AWK Programming Language, Addison-Wesley, 1988.
- [3] Mark Burgess, GNU Cfengine, <http://www.iu.hioslo.no/cfengine>.
- [4] Alva Couch and Michael Gilfix, <http://www.eecs.tufts.edu/~couch/prolog/>.
- [5] Ethan Galstad, NetSaint, <http://www.netsaint.org>.
- [6] Mark Lutz, Programming Python, O'Reilly & Associates, 1996.
- [7] Sean MacGuire, Big Brother, <http://bb4.com>.
- [8] Tivoli, <http://www.tivoli.com/>.
- [9] Jim Trocki, Mon (Service Monitoring Daemon), <http://www.kernel.org/software/mon>.
- [10] Unicenter TNG, <http://www.ca.com/>.
- [11] Larry Wall, Tom Christiansen, Randal L. Schwartz, Programming Perl, O'Reilly & Associates, Inc., 1996.

Appendix 1: piktc Command Options

```

Usage:  piktc <-cCdefGhikKlLm#rRstTvxxX>
        [+|-D          <define(s)>]
        [+C            <command(s)>]
        [+|-A|S[f]  all| <alert(s)/script(s)>]
        [+|-P[f]    all| <program(s)>]
        [+|-F[f]    all| <file(s)>]
        [+|-O[f]    all| <object(s)>]
        +|-H        all| <host(s)>
        [ALL]

-c          syntax check all config files
-C          syntax doublecheck all config files
-d          disable alert(s)
-e          enable alert(s)
-f          diff alert/script/program/file/
            object file(s)
-G          run in debug mode
-h          show program help
-i          install alert/script/program/file/
            object file(s)
-k          kill alert daemon (piktd)
-K          kill service daemon (piktc_svc)
-l          list alert/script/program/file/
            object file(s)
-L          list alert/script/program/file/
            object or host/os/group/alias
            command-line item(s)
-m#         checksum alert/script/program/file/
            object file(s), where # is
            checksum level 1-5
-r          (re)start alert daemon (piktd)
-R          (re)start service daemon (piktc_svc)
-s          show alert(s) status
-t          delete alert/script/program/file/
            object/history/log file(s)
-T          run in test mode
-v          run in verbose mode
-x          execute alert(s)/script(s)
-X          execute alert(s)/script(s)
            with no wait
+|-D        <define(s)>  define/undefine define(s)
+C          <command(s)> include command string(s)
+|-A|S[f]  all| <alert(s)> include/exclude (fix) alert(s)
+|-P[f]    all| <program(s)> include/exclude (fix) program
            file(s)
+|-F[f]    all| <file(s)>  include/exclude (fix) other
            file(s)
+|-O[f]    all| <object(s)> include/exclude (fix) object
            file(s)
+|-H        all| <host(s)> include/exclude host(s)/os(es)/
            group(s)/alias(es)
ALL         +A all +S all +P all +F all
            +O all +H all

```


Relieving the Burden of System Administration through Support Automation

Allan Miller & Alex Donnini – HandsFree Networks

ABSTRACT

The number of computer users with little or no training continues to rapidly increase. Networks are at the heart of companies large and small. Applications, ever more complex, span intranets and extranets. This all adds up to an increasing burden on system administrators and end user support organizations at a time when there is constant downward pressure on support budgets and a shortage of qualified staff. The result is a technical support crisis with dissatisfied end users, burned out system administrators, and unhappy support teams. The need for support automation is critical as it enables the scaling of effective end user support while minimizing the need for additional resources.

There are significant challenges to automating support, since it is essentially a large collection of special cases. Different methods have been used to achieve this automation, with varying degrees of success. This paper describes a software system that automates the solution of many recurring end user problems, greatly relieving the burden on system administration staff for mundane issues. We describe the architecture of the system, give examples of its use, demonstrate its extensibility, and report on our experience using it in the field.

Introduction

The widespread use of computers in mission-critical functions continues to grow, resulting in their use by ever-larger numbers of technically unsophisticated end users. At the same time, the software installed on them gets more complex as competitive pressures force vendors to incorporate additional advanced features. To make matters worse, the infrastructure itself is becoming inherently more sophisticated: a seemingly simple application such as email involves the use of network connectivity and routing that is far beyond the capabilities of an end user to diagnose and repair. Yet budgets typically do not leave room for additional support needs, as demonstrated in [Tol92]. The increasing demand for support services by end users puts system administrators in a position where they have to solve more and more repetitive issues, in addition to dealing with the more complex enterprise-wide issues they face every day. Often, they are simply not in a position where they can handle this additional burden.

Outside the enterprise, the situation is even worse. With less access to system administration resources, smaller businesses rely on a combination of internal resources and third party service providers. The labor intensive support process used by third party service providers simply does not scale to the level being demanded by current and future use. As evidenced in [DeK99], smaller businesses that cannot afford a full-time system administrator are increasingly dissatisfied with their support. As software becomes embedded in more and more consumer

products (such as cell phones or even refrigerators), the problem continues to worsen and is beginning to affect home users as well as businesses.

As a result, there is an urgent need for an increased level of automation in the process of everyday administration, maintenance, and support tasks for end users. System administrators must be freed from the overwhelming load of essentially trivial problems that recur on a frequent enough basis to interfere with the solution of more important and challenging problems. This paper describes a software system that automates the solution of many recurring end user problems, greatly relieving the burden on system administration staff for mundane issues.

Existing Support Automation

One of the indications of the need for automation of end user support is the existence of a range of products in this space. These products all address the need to some point, but have various shortcomings that are addressed by the system described in this paper.

There are a number of products that detect system level events, such as processor faults, and attempt to recover from their effects. These products use a general mechanism and apply it to all instances of the problems being detected. However, the general mechanism really only addresses the immediate cause of the problem event, and does nothing to permanently address the root cause of the underlying problem.

Some products provide automated software updates to customers. This solves problems caused by bugs that are fixed in a later release of the software.

However, many end user issues are not caused by bugs, but are instead usability issues where the software is working as designed but the designers have not foreseen its obscure behavior in a common circumstance. In addition, most problems can be addressed with workarounds that end users can apply immediately, eliminating the need to wait for the next turn of the software development cycle of the product.

Virus scanners are important tools for every system administrator. They detect system level events and changes to files, use a database to drive the detection and removal of viruses, provide reporting functions on virus detection and elimination, and periodically update the database automatically. As we will see, the system described in this paper is similar in some respects to a virus scanner. However, virus scanners have a very specific problem domain on which they focus. They provide no help to a system administrator for the myriad of other end user issues that arise on a daily basis.

“Self-healing” systems provide a facility, either manually guided or semi-automated, to restore some or all files back to a previous state. This can be a powerful device for quickly getting a system back to a working condition. However, it is most helpful in a disaster recovery situation, and is not of much help when a problem must be solved rather than removed, or when a problem has existed for a long time without manifesting itself.

Some “web-enabled support” sites provide a database of known problems and solutions (usually called a “knowledge base”). [Mic00] is a good example of a well-built knowledge base. While knowledge

bases have demonstrated their value in providing quick access to vital information for system administrators, the concept of their direct use by end users leaves much to be desired. Most end users have difficulty finding and understanding information in a knowledge base, and may actually end up doing more harm than good by attempting to apply inappropriate solutions. In addition, moving the task of system management into the hands of an inexperienced end user is not a very popular option in an enterprise environment where “time is money.”

Help desk software is a key component of an enterprise solution for end user support. Effective help desk software can increase the efficiency of an internal support group and therefore reduce some of the burden on system administrators. In addition, some help desk software can assist an internal support group in building a database of problems and solutions, which can be extremely useful to administrators. However, the software is of little direct use to those system administrators by itself.

Some help desk add-ons allow end users to contact a support group electronically, using email or chat, and facilitate problem resolution through the use of advanced diagnostics. Some of these products also feature limited automation of problem resolution activities and remote support capability, allowing a support representative to take control of an end user's desktop to resolve a problem. This removes some of the potential for human error in diagnosing and resolving the problem. These tools can certainly help a system administrator maintain a larger number of end users more efficiently. However, even with these tools, each end user issue requires individual attention, so

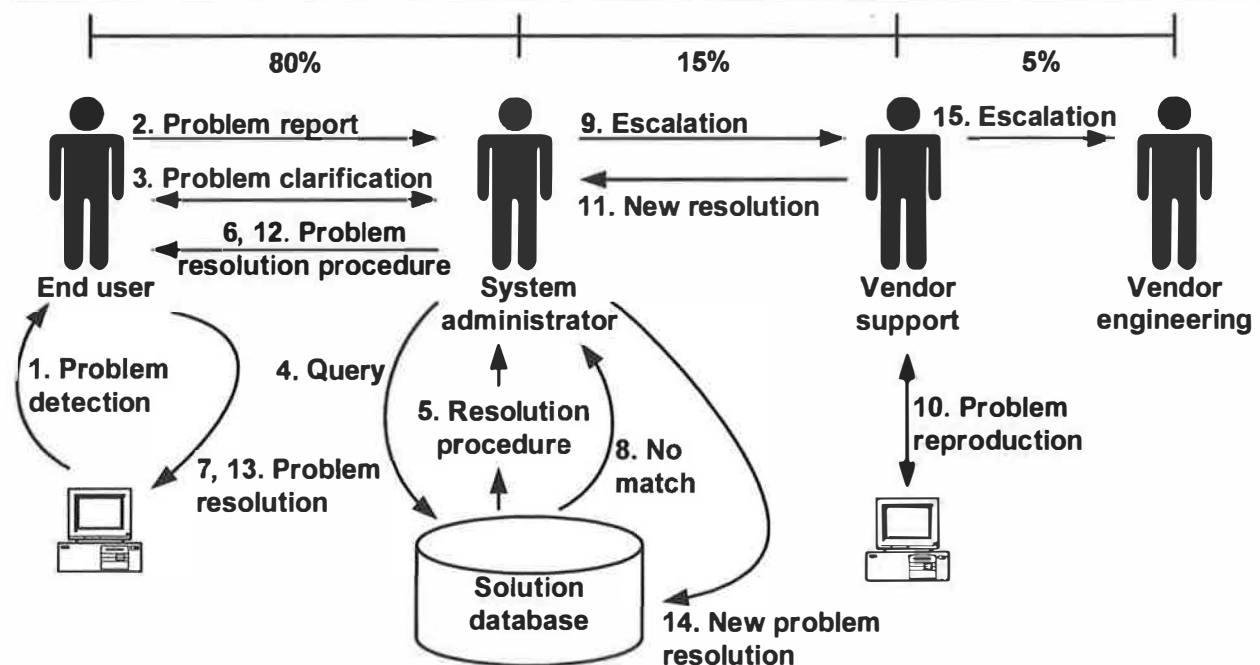


Figure 1: Support process.

the system administrator is still forced into the inefficient role of providing repetitive end user support on common problems.

"Expert marketplaces" match up end users with consultants who are bidding services to solve desktop computer problems. This is an interesting "free market" approach to providing end user support by outsourcing it to a widely distributed support staff. However, it does suffer from somewhat inconsistent results, due to the variety of talent bidding the support services. It is easy to imagine a situation where multiple consultants would inadvertently provide conflicting point solutions to a system wide problem, merely interfering with the efforts of the system administrator who is ultimately responsible for the overall solution.

Finally, a number of "support portals" are available that feature online access to some combination of the services listed above. These provide convenient "one stop shopping" for those services, but do not otherwise enhance the automation or scalability of the services themselves, and are typically geared to a lower level of expertise than that of a system administrator.

In summary, while these approaches are all valid and help with some aspect of the end user support issues faced by system administrators, none of them really shorten the administrator's lengthy task list by automatically detecting and resolving simple recurring problems without any manual intervention. The system described in this paper addresses many of these shortcomings and provides a true solution for saving time and lightening the load on the system administrator.

Automating Support

Figure 1 shows the traditional process for handling end user problems, as exemplified in [RSA00] and [Smi97]. The process starts when an end user determines that a problem is happening. The user contacts¹ the system administrator. Through a discussion with the user, the administrator clarifies the nature of the problem, then searches through a database of pre-defined problem descriptions for a match. In many cases, the "database" is simply a mental list of common problems and solutions. It can also be an informal written list, or if the system administrator is using help desk software to manage the process, it may be an actual database. If a match is found, the administrator communicates the resolution to the user, and assists the user in applying it. [Etc98] describes that this process works well, solving as many as 80% of the problems encountered by end users [Sla00, Gil00, Sup00, Sch00, Hon00, CWS99, Loc00]. For the remaining 20% or so, the system administrator must become directly involved in the

¹This could be a personal contact or a telephone call. Increasingly, other forms of interactive electronic communications are being used.

diagnosis and solution of the problem, and may need to escalate the process to include the software vendor. If the problem is escalated, the vendor uses product-specific knowledge to reproduce the problem and attempt to resolve it. If the resolution is successful, the procedure for resolving the problem is communicated back to the system administrator, who in turn communicates the resolution to the end user and assists in its application. In addition, the system administrator now adds this new problem and resolution to whatever "database" is used for solving common problems. In that way, new instances of the problem can be resolved more quickly, without needing either direct involvement or the assistance of the vendor. The 5% or so of problems that cannot be resolved by the vendor's customer support are escalated internally to the engineering team at the vendor, where development resources are brought to bear on diagnosing and resolving the problem. In this way, the efforts of the system administrator and the vendor's customer support group play two critical roles in the process: keeping the "database" populated with relevant problems and solutions for common recurring problems, and providing qualified, valid bug reports to the vendor's engineering staff.

Clearly, the amount of human effort involved in Figure 1 makes it a very labor-intensive and error-prone process. To make things worse, the "database" in Figure 1 may be simply mental notes, so the system administrator may be the only person who can drive problem resolution. Even with a more formalized process, the quality of the response is very dependent on the ability of the person using the database, whatever its implementation.

Figure 2 shows the process for automated support. A software client manages the process for the end user. When the client detects that a problem is happening, as described later, it searches a local database for a match to the symptoms of the problem. If a match is found, the database contains executable code that the client runs in order to resolve the problem. Since the database is functionally equivalent to the database in Figure 1, this solves as many as 80% of the problems encountered by the user. For the remaining 20% or so, the client forwards a detailed description of the circumstances surrounding the problem to the system administrator, who is made aware of the problem early on. At that point, the administrator may choose to solve the problem or may instead want to escalate it to the vendor. In either case, the information from the client can be used to diagnose, reproduce, and solve the problem. When a solution is found, either the system administrator or the vendor codes and tests a solution for the problem and adds it to the master database. This master database is used to update the local database, which the client then uses to resolve the problem. The 5% or so of problems that cannot be resolved by the system administrator or vendor's customer support group are escalated to the

vendor's engineering staff. The system administrator and vendor's support group continue to play the same two critical roles as in Figure 1: keeping the master database populated with relevant problems and solutions for common recurring problems, and providing qualified, valid bug reports to the vendor's engineering staff.

It is worth pointing out that some problems cannot be detected, and some solutions cannot be implemented, by the system outlined in Figure 2. The client cannot automatically detect problems that completely disable the system, such as a faulty power supply, since the system cannot run the client. The client cannot automatically implement a solution that requires physical modification of the system, such as replacing memory. Likewise, the client cannot automatically detect problems that are not amenable to software detection, such as poor quality in printer output. Finally, the client cannot automatically resolve problems that are caused by a misunderstanding on an end user's part, such as inability to set the margins in a word processor. However, two powerful techniques can be used to overcome some of these limitations. The first is the use of communications between clients on multiple machines. Even if a problem cannot be detected directly on the machine where it occurs, its external effects may be readily identified on other machines in the network. The second is the use of an interactive dialog with the user of the machine. Even when the client cannot test or modify certain aspects of its environment, it may be able to instruct the user to do so on its behalf. In the end, even if the client cannot completely diagnose or resolve a problem, the

information it gathers in doing so will be immensely helpful to the system administrator ultimately responsible for the problem resolution. Our analysis, described later, of the most frequently occurring problems in the Microsoft Knowledge Base, [Mic00], indicate that software configuration issues cause the vast majority of these problems. This means that the kinds of problems the client cannot address make up a relatively unimportant fraction of those that actually happen. Anecdotal evidence also seems to support this conclusion.

Two important features of automated support are evident in Figure 2. The first is that up to 80% of the problems that occur on the end user system are resolved quickly and accurately, without any human intervention. The user may not even be aware that the problem existed in the first place. The second is that the tasks of the system administrator are driven by the appearance of new problems, not by the appearance of new users. This means that the part of Figure 2 that needs to scale up the most to handle more end users is the update of the local database from the master database. An increase in the number of users does not require a proportional increase in the system administration group, so large numbers of users can be supported without a huge staff.

There are significant challenges to automating support. By definition, the task is a large collection of special cases, but software tends to work best with small sets of algorithms applied to large numbers of uniform tasks. In addition, the applications that are being supported cannot implement the support themselves, since they are presumably malfunctioning.

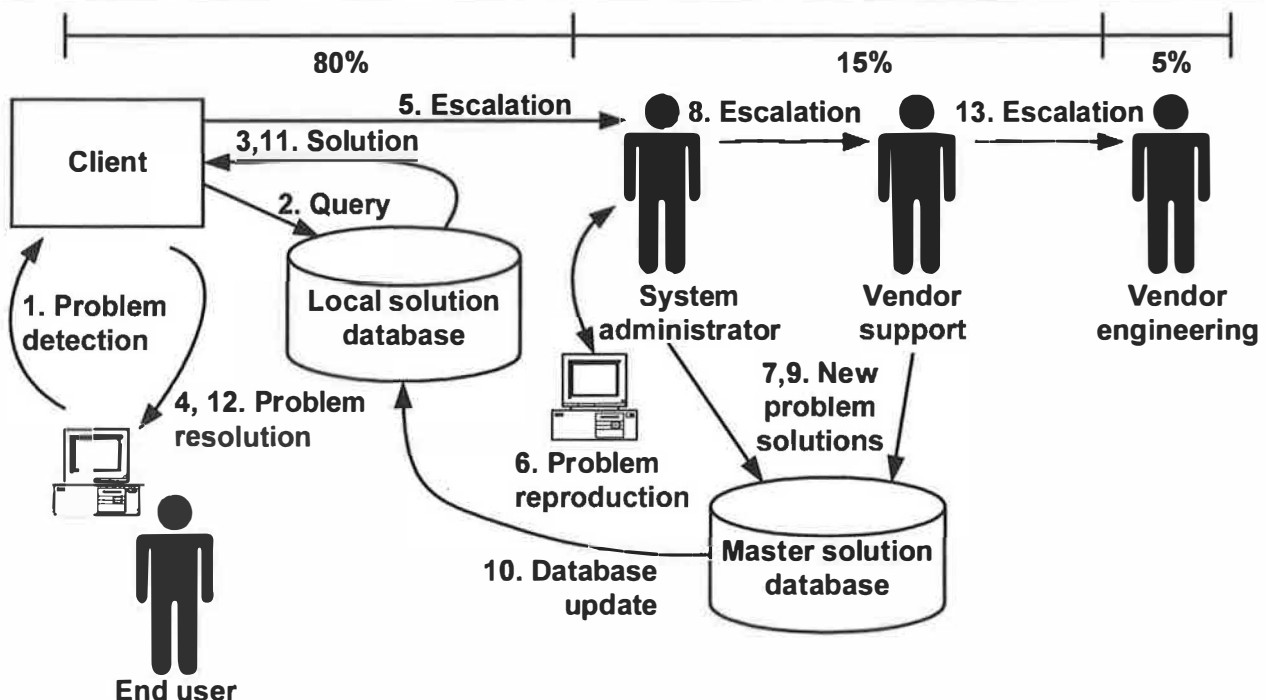


Figure 2: Automated support.

Since nearly any aspect of system operation can be part of the problem, and part of the solution, the automated support tool must be as independent as possible from the system and its applications. At the same time, it must be aware of most of the details of the system operation.

These factors make the job of automating support difficult, but not impossible. Database techniques can be used to manage the large number of special cases. Both the diagnosis of problems (“what happens when you run...?”) and their resolution (“now change the setting...”) are essentially software tasks that can be automated. Combining these two considerations, the implementation of automated support works well with a database containing executable code for the diagnosis and resolution of problems. The size of the database is strongly affected by the “80-20 rule”: 80% of end user issues that arise can be solved using only 20% of the entire universe of solutions. (Note that this is a little different from the 80% and 20% mentioned above, which referred to the fact that 80% of user issues can be resolved using a database.) The 80-20 rule is well known in support circles [Che99, Che99a, Gia99, Lan00, Ste99, Sum98]. The implication is that a relatively small database can be extremely effective in resolving user issues. We now describe the use of such a database in an automated support system provided by HandsFree Networks.

System Architecture

The heart of the system is the client that runs on each end user machine in the facility. Figure 3 shows an overview of the client. The database contains a collection of scrips,² each of which contains executable code to diagnose and resolve a single problem. A scrip consists of four major parts: initialization, configuration, symptom, and solution. These major parts are described in more detail later in this paper.

²The term “scrip” is borrowed from the medical profession, where it is the working jargon for a prescription. Like these database entries, pharmaceuticals are used both for diagnosing and resolving problems. There is also a connection to the word “script.”

Event detection drives the application of scrips to problems based on system level events such as window creation, user input, processor faults, process creation and termination, device notifications, and timer expiration. The event detection module efficiently determines which, if any, scrips should be run as a result of a sequence of system level events. It is designed to very quickly dismiss events that do not trigger a scrip. Since the resulting conceptual model of a scrip is an interrupt service routine rather than a polled service, a scrip only uses system resources when a problem is detected, thereby minimizing its impact on the system.

The execution scheduler is responsible for dynamically loading and executing the code for a scrip on an as-needed basis. It is in charge of managing the synchronization of scrips that are running in multiple threads, as well as passing information from detected events to the scrips. The database also contains procedures that are dynamically loaded when they are called from a scrip; the execution scheduler manages the loading and execution of these procedures, as well as the passing of parameters into and out of the procedures.

Primitives are a library of pre-defined routines that provide a convenient (and, where possible, system independent) interface to operating system functions. They also provide an interface to common utilities such as memory allocation and string manipulation. Scrips and procedures in the database use primitives to accomplish most of their work. From the perspective of a scrip, the interface to a procedure is the same as the interface to a primitive, so procedures in the database serve as a convenient and flexible way to extend the primitives.

Scrips use the persistent state mechanism to keep information that must persist across multiple invocations of the scrip and machine restarts, as well as information that must be accessed by more than one scrip. Variables in persistent state have a namespace that limits their scope, so a scrip author can freely

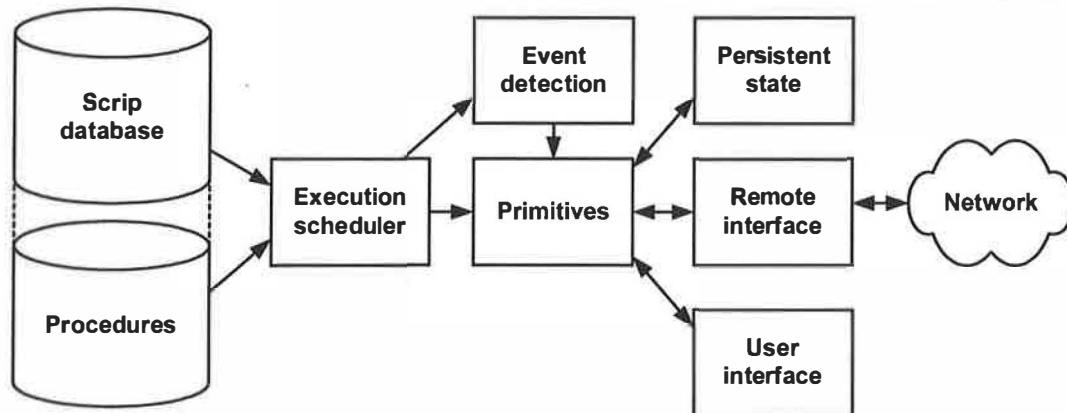


Figure 3: Client Architecture.

define and use a persistent state variable without concern for interfering with the variables in other scripts.

The remote interface allows scripts to run code with equal ease on any machine in the network, including the machine on which the script is currently running. This is accomplished by making every call to a primitive specify the machine where it should run. An example of this is described in more detail later in this paper.

A key feature of the client is its portability to multiple environments. It has been designed from its inception to have well-defined system dependent and system independent modules with consistent interfaces. One of the authors has architected three major portable software systems in the past (described in [Vir97, Ter00, CAS87]), and has brought relevant experience to bear on this system as well. The current version of the client runs on many implementations of the Win32 API (Windows 95, Windows 98, Windows NT 4, Windows 2000, and Windows Me), and a Linux version is presently in development.

Much of the value of the system comes from the fact that the scrip database is updated on a regular basis, since end user issues change regularly as new products are introduced and new problems are discovered. As a matter of fact, one of the greatest benefits of implementing the client using a database architecture is the flexibility in providing this update. For example, the scrip database can be implemented as a central database, or a central database with local caching, or a distributed database, or a redundant database, or any combination of those options. In this application, a problem that affects network connectivity may isolate the client from the network, so a standalone local database is desirable. A simplified representation of the default product configuration is shown in Figure 4. A centralized master scrip database

contains all available scripts. At the system administrator's facility, a local scrip database stores the subset of the master scrip database that is relevant to the local hardware and software configuration. At a regular interval (about once a week), the client on one of the administration machines initiates an incremental update of the local database to retrieve any new or modified scripts. All the machines at the facility access the local scrip database for the latest versions of all scripts. In addition, they also have a small database of scripts to solve connectivity that is replicated on every machine. If a machine cannot access the local scrip database due to a connectivity issue, this database helps to solve that problem first. Note that both the master scrip database and local scrip database can be replicated for improvements in reliability and performance.

The implementation of security in the system is of paramount importance. Since the primary function of the software is to transmit and run executable code, it would be an ideal virus infection vector. To prevent this sort of abuse, all network access goes through the remote interface and uses the HTTP protocol. The remote interface implements SSL as described in [Fre96]. For maximum security, it can be configured to require both client and server authentication. This encryption protects system integrity by preventing malicious attacks on the code executed by a client, and also protects data security by preventing passive observation of the data communicated between two clients while executing a primitive remotely. In addition, all scripts are digitally signed, and the engine can be configured to only run scripts from a list of trusted providers. Even without strong (or any) encryption, the digital signatures prevent the execution of malicious code by a client, since the attacker can forge a script but cannot forge its signature. This protection works for worldwide deployment of the database,

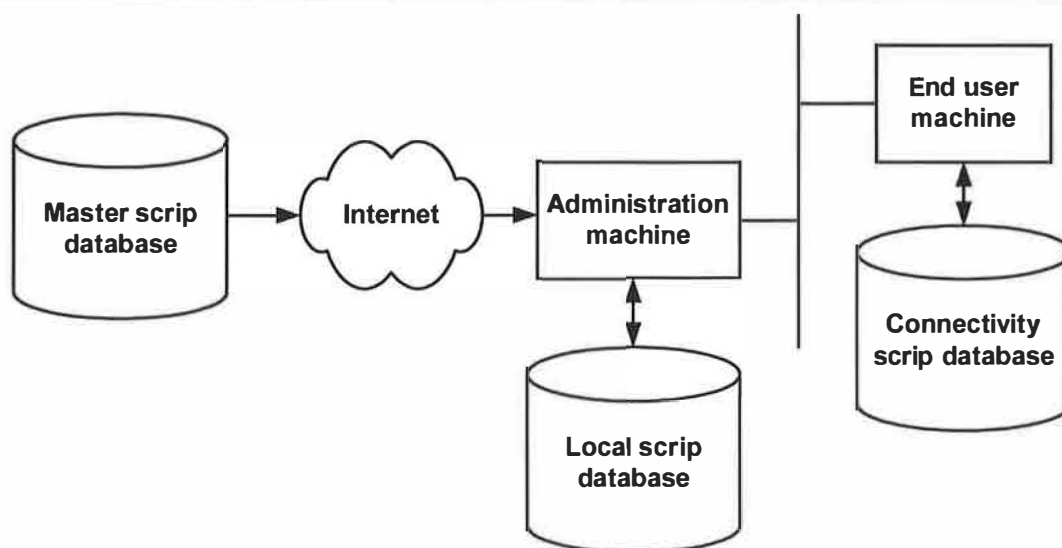


Figure 4: Scrip databases.

since export restrictions on strong security only apply to data encryption, not to digital signatures.

Interestingly enough, the connotation of a “certificate authority” for the SSL security in the client is somewhat different from the normal one. In secure e-commerce transactions, a certificate authority is simply declaring that the identity of the entity being certified is actually correct. A certificate authority for scripts, on the other hand, is declaring that the entity being certified is one that can be trusted to produce reasonably stable and desirable scripts. Presumably, the certificate authority provides this service by verifying that the entity has reasonable QA procedures in place, doing periodic audits of the software production facility, and doing random independent code reviews and testing of the scripts themselves. This is quite a bit more complicated than simply verifying identity, but is worth correspondingly more as a value added service. HandsFree Networks plans to provide this service as part of its product offering, but also expects that third party providers will provide a similar service.

Problem Resolution Process

Referring to Figure 2, the problem resolution process starts when the event detection mechanism recognizes a certain sequence of system level events that corresponds to an entry in the scrip database. It puts together relevant information about the system level events into an “event” that is passed to the execution scheduler, along with an identifier indicating which scrip is being triggered.

The execution scheduler decides when the triggered scrip should be run and loads the scrip symptom from the database. It runs the code for the symptom, which does any kind of verification needed beyond the trigger to ensure that the problem is indeed present. If the symptom indicates that the problem is present, the execution scheduler loads the scrip solution from the database and runs the code for it.

When a problem cannot be resolved locally, an escalation process prevents the situation where an end user is left without a solution. Escalation can be initiated by the failure of a solution, which is detected when a scrip continues to diagnose a problem even after its resolution has been applied. Escalation can also be initiated by a “diagnostic” scrip, which detects a general error condition that does not have a specific solution supplied by another scrip. A set of about twenty diagnostic scripts is used to detect errors that are not resolved by the rest of the scripts. The first step of the escalation process is to initiate an update of the local database from the master database. In the most common case, the problem is a relatively new one that has been discovered and resolved since the last periodic update of the local database. For example, it might be a compatibility problem caused by installing a new version of a popular program. In this

case, the new scrip to resolve the problem is retrieved and applied. If, on the other hand, the problem is being encountered for the first time, the second step of the escalation process is to gather relevant information about the state of the system and the error and forward it electronically to the system administrator. This is typically much more complete information than the system administrator usually gets from end users. The administrator can then either reproduce the problem locally to determine its resolution, escalate the problem to the vendor, or simply contact the user directly and resolve the problem through more traditional means. If the user is not local, the system administrator can initiate a “remote service” session, allowing the use of the remote interface module of the client to look at specific system state and apply problem resolutions. After understanding and resolving the problem, either the system administrator or the vendor’s customer support staff initiates a process to add a new scrip to the master database. If the problem cannot be resolved and is escalated to the vendor’s engineering staff, it follows the same support process that it would for non-automated support. Once a resolution is available, however, the system administrator or vendor adds a new scrip with the resolution to the master database. In any case, once the new scrip is in the database, the same problem is always resolved on all end user machines without further manual intervention.

The software also provides a valuable service for administrators who are managing multiple end user sites by providing notification of all detection and resolution activities. This can be used simply for accounting purposes, so that the administrator can easily provide activity reports to management demonstrating the ongoing value of the system administration staff. It can also be used for isolating trends that indicate a particularly troublesome hardware or software component at a particular site, leading to a replacement of that component and a resulting improvement in stability. Finally, it can be used to provide greater visibility to the system administrator. For example, the decision on when to upgrade an application to a new version is typically made on a fairly arbitrary basis today. However, with this notification, the administrator is aware of the number of end user problems occurring that could be solved by such an upgrade. If the number of such problems is small, the upgrade can be postponed, but if it is large, the upgrade might be accelerated. Since most users rarely, if ever, report this kind of information, it is extremely valuable to system administrators. Detailed reporting of problem detection and resolution activities also gives system administrators leverage when dealing with vendors’ customer support groups, since it minimizes the frustrating “finger-pointing” that typically goes on when problems occur in the multi-vendor computing environments that make up nearly all facilities.

This reporting mechanism is very configurable. It can be set up on a scrip-by-scrip basis to report immediately or collect the reports into a single log that is sent on a periodic basis. An email interface provides the simplest, most convenient means of reporting. For system administrators who are maintaining a larger facility, a web-based interface can log reports by adding entries to a database. The administrator can then use a browser interface to peruse the database, spot trends, and manage user issues remotely.

Scripts

Each scrip consists of four parts: initialization, configuration, symptom, and solution. The symptom and solution parts have already been described; both parts are executable code that is loaded dynamically and run by the execution scheduler. The code for the symptom does a conclusive test to see whether the conditions for the problem actually exist. This is required since the events triggering a scrip may not completely define the situation for the problem. For example, the trigger may be an error dialog, but it may be necessary to look at some configuration data (in files or the registry) to determine whether the known problem is actually happening. The code for the solution is the actual implementation of the solution. It can be extremely simple, for example, a solution that solely involves updating some configuration information. It can also be quite complex, for example, a solution that has to try several different resolutions to see if any of them work, and must restart the system (and coordinate this process with the end user) after each try.

Since the execution scheduler needs to set up the interface between scripts and primitives (and also between scripts and procedures) at runtime, the procedure interface across the boundaries of scripts must go through a thin interface layer that “interprets” the calls and returns. This has an impact on the performance of scrip execution; however, scrip execution happens at a low frequency and is not time critical, so the overall impact is negligible. On the other hand, one large benefit of this architecture is that symptoms and solutions for scripts can be implemented in any language for which it is possible to implement this thin interface layer. Current implementations of scripts use the “C” programming language, but this is mostly a matter of convenience, and interfaces for popular programming languages such as Perl (see [Wal00]) are being developed.

The initialization part of a scrip has several functions. It registers the scrip with the execution scheduler, specifying characteristics about the scrip such as other scripts with which it must be mutually exclusive (cannot run at the same time), and other scripts upon which it depends. Most scripts can work with a default configuration, but more complex ones can have specialized requirements. Initialization of a scrip also defines the persistent state variables used by the scrip.

Finally, the initialization defines the conditions under which the scrip is triggered. This is done by specifying the conditions on a sequence of a few fundamental events. For example, a scrip can register itself to be triggered when a certain executable is run with a specific command line, and then a dialog box is created by the resulting process with a given title.

The timing and order of scrip initialization is left unspecified. For a client that needs to implement completely dynamic configuration, the scrip initializations can be run sequentially as the client is starting. For a more conventional client (such as one used in desktop support), the scrip initializations can be run as a pre-processing step whenever the client is installed or the scrip database is updated. The results of the initialization are saved in runtime tables that are read during client startup to provide fast initialization.

The configuration part of a scrip allows the scrip author to create user-defined options that control the function of the scrip. This section specifies classes of input fields, such as radio buttons, check boxes, and text strings. Each input field has a default value and a persistent state variable associated with it. The user interface module presents the configuration information as XML, in a form that can be displayed and modified by a browser. Once this setup is complete, the client uses it to initialize the referenced persistent state variables. When the scrip runs, it uses the values of those variables to control its operation. In this way, the scrip author has great flexibility in allowing customization of the operation of the system. This facility also allows customization of “overall” features of the system, since the persistent state variables that a scrip configuration modifies are not limited in scope to variables that affect a single scrip.

There are actually four general types of scripts, although these types are all implemented the same way, so the categorization is a conceptual one rather than an operational one. The scripts that have been emphasized so far are those that solve a specific known problem with a well-defined and tested solution. Another type of scrip is one that detects and resolves problems that arise as a result of end user operations, such as installing a new piece of hardware or software. Still another type of scrip is the “diagnostic” scrip previously mentioned that detects a general error condition that does not have a specific solution supplied by another scrip. The fourth type of scrip is the “preventive” one that performs routine administration and maintenance functions that can be automated. Clearly, this last type of scrip is immediately useful to a system administrator, even in a system that is functioning perfectly.

To simplify the management of scripts, and to allow them to be conceptually classified, a tree-structured descriptive hierarchy is maintained. This is similar in concept to the hierarchy that is maintained for information accessed by a search engine such as

Yahoo! (see [Yah00]). The hierarchy is expanded as needed when new scrips are developed, and a single scrip can be in more than one place in the hierarchy. The hierarchy is represented in the database containing the scrips, and the user interface provides a facility for navigating the hierarchy to get to the configuration of individual scrips within it. Groups of scrips within a single branch of the hierarchy can be managed as a unit. For example, there are a group of scrips that implement basic functionality within the system, such as sending periodic logs and updating the client software to newer versions. These scrips are required for proper system operation, and are all part of the group of "system" scrips within the hierarchy. Implementing much of the system functionality using scrips makes it convenient to update the basic operating functions of the system without requiring software upgrades in the core software, resulting in a more reliable and flexible system.

One of the key design goals in the architecture of the scrip database is to allow authoring of scrips at a multitude of facilities without any explicit coordination. In this way, local expertise with certain types of problems can be captured and disseminated without having a central authority as a bottleneck. The namespace for persistent state, along with the self-organizing features of the scrip initialization and the descriptive hierarchy, combine with the modular nature of the scrips themselves to make the promise of this widely distributed development a reality.

Example

A concrete example of a scrip will help to understand many facets of the system. This example scrip solves a problem that is taken from the direct experience of one of the authors, a problem that occurred on a regular basis and was extremely annoying.

The underlying cause of the problem lies in the shortcomings of the Microsoft Windows print server architecture. It would not arise in a purely Unix environment, but most system administrators recognize the necessity of managing systems with the popular Windows operating system, and certainly concur with the desire to quickly and efficiently handle the many problems it introduces.

This particular problem manifests itself in this way: the facility has a Hewlett Packard LaserJet 4 printer, driven by one of the file server systems running Windows. Whenever a new system is installed, the printer is added as a network printer. Usually, everything works correctly, but occasionally someone prints a document and gets a dialog reporting that the document is "too complex" to print. However, if the document is transferred to another machine and printed from there, it prints with no problems.

This problem is acutely frustrating for the end user, because there is no apparent difference in the two systems (same printer driver, same document, same

operating system), yet the operation works correctly on one system but not on the other. In addition, the workaround of printing the document from another system is extremely inconvenient, because it disrupts the workflow of two people: the person trying to print the document and the person whose system must be used in order to print it. As a result, the person with the problem document usually ends up trying to "tweak" it so that it isn't "too complex" to print, resulting in wasted time and an inferior document.

The cause of this problem is the fact that the printer drivers for the HP LaserJet 4 are installed with a default configuration of 2 MB of memory in the printer, even if the printer has more memory. This configuration must be manually changed to reflect the actual amount of memory in the printer, or else the printer driver will not be able to format the document to print because of this perceived lack of memory. In addition, the printer driver has a feature called "page protection" that must be turned on in order for the driver to accurately estimate the memory that is needed in order to print a particular document. Unfortunately, these two configuration changes must be made every time the printer driver is installed. Since end users may be installing these drivers themselves, it is difficult to set up a process that insures the application of these configuration changes. The situation is further complicated by the fact that a long period of time may elapse between the installation of the driver and the appearance of the problem. Justifiably, the reaction of the end user is that everything has worked perfectly for a long period of time, so there must not be any problem with the printer setup.

This problem is ideal for solution with a simple scrip. Doing so provides a systematic way to represent "institutional" knowledge that is typically maintained informally by the system administration staff. The problem has a relatively simple method for recognizing it, a fairly straightforward means to test for its existence, and a clear procedure for solving it. It is interesting to note that this problem, like many that are addressed by system administrators, is not technically a "bug", in the sense that the software is working as designed. It is really more of a usability issue.

The scrip for this example will be presented using "C" as a programming language. The details to translate it to any particular programming language are fairly mechanical. The scrip initialization is shown in Listing 1. The initialization starts by registering the scrip with the execution scheduler, giving it a title and the default execution mode, which indicates that only one instance of the scrip should be run at a time. Then it sets up a trigger with the event detection mechanism, indicating that the scrip should be run whenever a dialog is created with the title "HP LaserJet 4" and the dialog text "Document too complex". Note that the first parameter for each primitive indicates the machine where the primitive should be run. In this case (and in all initialization), the primitives are run

on the current machine, which is indicated by the constant CUR.

The symptom for this scrip is shown in Listing 2. The symptom gets the value of a specific registry key that contains the settings for the printer driver. If the registry key does not exist, then the scrip does not apply. It checks to see if the settings are the default ones for the driver, which are represented by a specific 310-byte binary value. If so, then this scrip applies, because it is designed to solve the problem where the default value has not been updated to match the actual properties of the printer. Note that once again, the primitives are all run on the current system. Also note that the parameter to the scrip is a data structure describing information about the circumstances triggering the scrip, but in this case that information is not needed to determine the applicability of the scrip.

The solution for this scrip is shown in Listing 3. The strategy for the solution is to assume that the printer driver settings are correct on the machine where the printer is installed, and copy the settings from that machine. It uses a registry key to get the name of the remote printer, then extracts the name of the machine. It converts the name into the internal representation of a machine identifier, then gets the

registry key for the printer settings. This is an example of executing a primitive on a remote machine. The same primitive is used to get the printer settings on the remote machine as was used to get them on the current machine in the symptom, but the first parameter indicates the remote machine rather than the current machine. Finally, the settings found on the remote machine are used to update the settings on the current machine. If at any point, the solution cannot proceed, it returns FALSE, indicating that it has failed. This failure will initiate the escalation process previously described. If the solution succeeds, then a successful problem solution will be logged as described previously.

Database Feasibility

One important question about the system that naturally arises concerns the feasibility of creating and maintaining the database of scrips. We have done significant research to address this question. As mentioned previously, the 80-20 rule indicates that the database size need only be about 20% of the total universe of problems being addressed. Data presented in [All99, Etc98, IHS00, ITS99, and Rea99] lead us to expect that 80% of all end user issues will be resolved

```
extern void Init(void)
{
    /* Register the scrip with the execution scheduler. Use the
       default setup: only one instance of this scrip at a time. */
    RegisterScrip(CUR,
        "Update HP LaserJet 4 setup to correct Document Too Complex".
        STANDALONE);
    /* Set up the trigger for the scrip. */
    SetDialogTrigger(CUR,
        NULL, /* owning process */
        "HP LaserJet 4", /* title */
        "Document too complex"); /* text */
}
```

Listing 1: Code for script initialization.

```
extern BOOLEAN Symptom(TRIGINFO trig)
{
    char *setupVal;
    BOOLEAN exists, compare;

    /* Check to see if the driver is installed, and get setup. */
    GetRegistryBinary(CUR, &setupVal, &exists,
        "HKEY_LOCAL_MACHINE\\System\\CurrentControlSet\\Control\\"
        "Print\\Printers\\HP LaserJet 4\\Default DevMode");
    /* If this driver isn't installed, the scrip doesn't apply. */
    if (!exists) return FALSE;
    /* See if it is still the default setup */
    CheckMemoryEqual(CUR, &compare, setupVal, DEFAULTVAL, 310);
    /* If the comparison is equal, the scrip applies. */
    return compare;
}
```

Listing 2: Symptom code.

by this database. We have extensively characterized the Microsoft Knowledge Base [Mic00], focusing on the major products including operating systems, office productivity applications, groupware, and Internet products. Using tools such as FAQs to identify the most frequently encountered problems, we found that a database with 2,500 to 3,000 entries should provide resolution of a large percentage of the recurring problems encountered by end users in a Microsoft Windows environment.

After the database is built, it needs to be maintained on an ongoing basis. Figure 5 shows the number of new entries as a function of time for the category "Windows 95" in [Mic00]. Except for an 8-month window around the operating system release, it shows about 25 new entries every month. Note that this is a pessimistic estimate, since not every entry in [Mic00] represents a problem, and even the "problem" entries do not all represent solutions that require automation. By way of comparison, Figure 6 shows the number of new entries as a function of time for the category "Windows networks" in [Mic00]. This graph has no particular peak around a product release, but shows about 10 new entries every month.

It seems reasonable to say that even an extremely pessimistic assumption for new problem incidence

will be less than 80 problems per month. Our initial work has shown that coding and testing a scrip takes less than four work days, so fewer than twenty full time scrip programmers can maintain the database.

In a way, virus scanners serve as an "existence proof" for the feasibility of the scrip database. The effort involved in obtaining, analyzing, and producing a solution for a virus is not too different from the effort involved in understanding a problem and creating a scrip for it. Current virus scanners have databases of about 45,000 viruses, so the effort involved in creating and maintaining the scrip database should be relatively small by comparison.

Field Experience

An initial version of the software described here is currently deployed and operational at a number of customer sites. It has been running at one site, a public relations firm with about 15 nodes, since February 2000. Starting in March 2000, major functionality enhancements were released to that site on a weekly basis, and in April 2000 installations began at additional customer sites. As of the time of this writing (September 2000), the software is running at ten customer sites.

```
extern BOOLEAN Solution(void)
{
    char *port;
    char *machineName;
    char *setupVal;
    MACHINE mID;
    BOOLEAN exists;
    /* Get the machine where the printer is installed. */
    GetRegistryString(CUR, &port, &exists,
        "HKEY_LOCAL_MACHINE\\System\\CurrentControlSet\\Control\\"
        "Print\\Printers\\HP LaserJet 4\\Port");
    if (!exists) return FALSE;
    if ((port[0] != '\\') || (port[1] != '\\')) return FALSE;
    GetSubstring(CUR, &machineName, port, 2, '\\');
    GetMachineID(CUR, mID, machineName);
    /* Get the printer setup from that machine. */
    GetRegistryBinary(mID, &setupVal, &exists,
        "HKEY_LOCAL_MACHINE\\System\\CurrentControlSet\\Control\\"
        "Print\\Printers\\HP LaserJet 4\\Default DevMode");
    if (!exists) return FALSE;
    /* Use that printer setup on this machine. */
    SetRegistryBinary(CUR, &setupVal,
        "HKEY_LOCAL_MACHINE\\System\\CurrentControlSet\\Control\\"
        "Print\\Printers\\HP LaserJet 4\\Default DevMode");
    /* Notify end user of required action. */
    NotifyUser(CUR,
        "Your printer setup has been updated to correct the "
        "'Document too complex' problem. Please try printing "
        "the document again.");
    return TRUE;
}
```

Listing 3: Solution code.

The customer feedback has been very positive. None of the customers have reported any problems or performance degradation with the software, and the software has proven its value in identifying major issues at every single one of the sites. The system administrator at one site was pleased to report that the logs have provided information on problems that would otherwise go unreported. At another site, the logs indicated a serious problem that was preventing the virus scanner from running, which had gone undetected for months!

The feedback from customer sites has also proved very valuable to our development team. The client checks every error return and uses its own reporting mechanism to log unexpected results. This process has unearthed several non-fatal, but

potentially serious, bugs that were not found during testing. In early versions of the client, the problem resolution capabilities have been limited, so every event was escalated through the reporting mechanism. These reports have guided us in our choice of scripts to implement for the initial database, so we are confident that we are populating the database with the most useful scripts. One reassuring "reality check" is the observation that the problems that occur at all ten customer sites are remarkably similar, even though the businesses themselves are considerably different.

Open Source Development

The scrip database lends itself very naturally to an open source development methodology. The initial commitment to the development of a scrip is small

Win 95 Knowledge Base

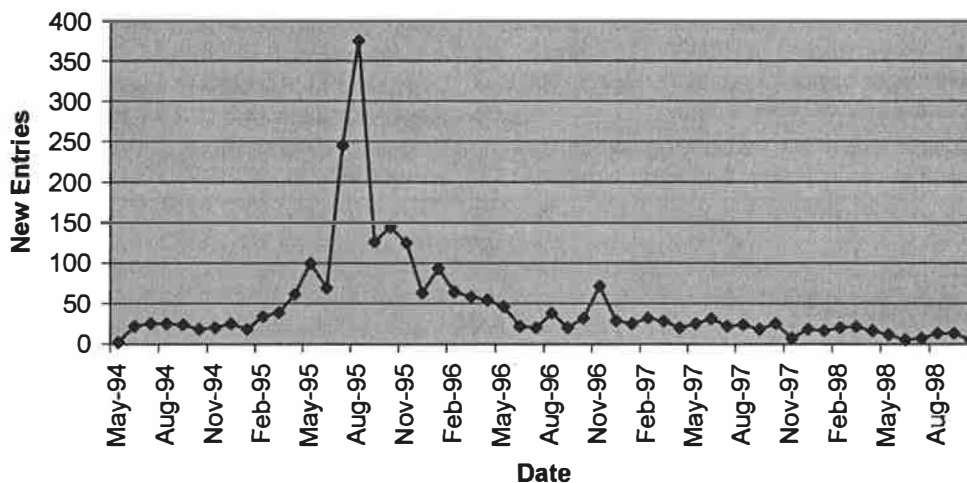


Figure 5: New Problem Incidence in Windows 95.

Win network Knowledge Base

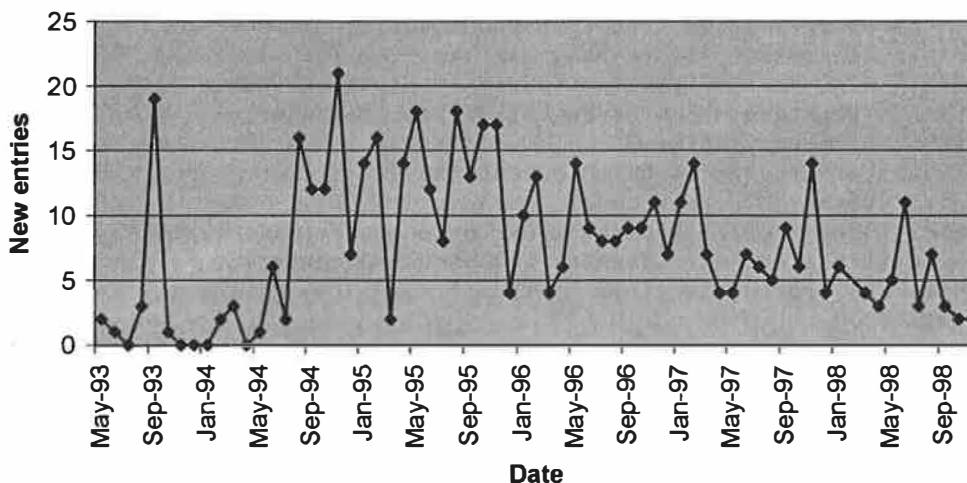


Figure 6: New Problem Incidence in Windows Networking.

since it only takes a few days, and the payback is immediate since it solves a persistent and annoying problem in a permanent way. Scripts are likely to be useful at more than one site, so the code that goes back into the open source pool will be immediately beneficial to the user community at large. This approach also helps to populate the database with the most important scripts first, since the motivation will be highest to solve the most severe problems. The open source methodology also adds a measure of security, since open review of the scripts will deter any malicious activity. A more subtle benefit arises from the fact that script development has several distinct phases, including problem resolution, solution definition, coding, and testing. These phases require different skills and can be completed most efficiently in a cooperative group environment. For all these reasons, we are using an open source methodology for script development.

Future Directions

Although this system is primarily focused on reducing system administration headaches by streamlining end user support, it lends itself naturally to several other uses that can also make an administrator's life easier. As previously described, we have written scripts both for bugs and for usability issues that are frequently encountered by end users and are confusing enough to generate calls for help. In addition, we have written scripts to automate common administration and maintenance functions that help to keep a properly functioning system running smoothly. We also see great utility in using scripts to distribute security patches, since the conditions for their use and the procedures for their application are often complex enough to warrant code for their implementation.

The technology is flexible and useful enough to apply to the next generation of "pervasive" computing devices. As the load on system administrators expands to include support of mobile and other embedded computing platforms, the need for automated support is only going to become more critical. The lightweight, standards-based client is ideally suited for these environments as well as the applications that support them.

Summary

The HandsFree Networks support automation tool described in this paper relieves system administrators of the burden of dealing with mundane issues by automating the resolution of many common recurring problems. It uses a standards-based extensible architecture to provide a database of solutions that is applied without manual intervention. An open source development methodology for the database and a built-in incremental database update ensures that the solutions will keep up with the introduction of new products and the corresponding appearance of new issues.

Author Information

Allan Miller received a B.S. in Electrical Engineering from Purdue University in 1979 and an M.S. in Computer Science from Stanford University in 1982. While on his way to a Ph.D. in Computer Science from Stanford, he left to start CASE Technology in 1983, to create and sell electronic CAD software. After CASE was bought by Teradyne in 1987, he left in 1993 to start Virtual Music Entertainment, providing a unique technology that allows non-musicians to enjoy playing music (the Virtual Music software is featured on Aerosmith's 1997 "Nine Lives" album). Allan likes to bicycle and play the piano, and he plays drums in the "well-known" Palo Alto band, The Wizards. He is active in town politics, serving on the Zoning Board of his Hollis, New Hampshire home town.

Alex Donnini has more than 20 years of experience in the information technology industry. During the past ten years he has acquired direct, extensive knowledge of the information technology challenges faced by small businesses and of their technical support needs. This led him, together with Allan Miller, to found HandsFree Networks. Their goal is simple: deliver the first automated support system initially targeting workgroup and departmental information systems. Throughout his career, Alex has specialized in getting products or companies off the ground. HandsFree Networks is his third start-up, the second one that he has co-founded. He received an Honors' B.S. in theoretical statistics from the University of Western Ontario in 1977, and a Master's degree in Business Administration from Harvard University in 1981.

References

- [All99] M. Allimadi, "Companies Deploy Multi-Networked Call Centers to Deliver Efficient Customer Service," Call Center, <http://www.callcentermagazine.com/article/CCM20000427S0016>, June 1999.
- [Bra98] T. Bray, J. Paoli, and C. Sperberg-McQueen, "Extensible Markup Language (XML) 1.0," <http://www.w3.org/TR/1998/REC-xml-19980210>, Feb. 1998.
- [CAS87] CASE Technology, Inc., *The CASE Design System*, marketing literature, 1987.
- [Che99] A. Chen, "Sinking support costs," *eWEEK*, <http://www.zdnet.com/eweek/stories/general/0,11011,409655,00.html>, July 1999.
- [Che99a] A. Cheng, "Resurrected Technology for the Web: Expert Bites; AI Systems in digestible chunks," *Software911 white paper*, register at <http://www.software911.com/form/default.asp>, 1999.
- [Com97] D. Comer and D. Stevens, *Internetworking with TCP/IP, Volume III: Client-Server Programming and Applications, Windows Sockets Version*, Prentice-Hall, <http://www.cs.purdue.edu/homes/comer/tcpip3w.cont.html>, 1997.

- [CWS99] California Child Welfare Services, *1999 Year in Review*, http://www.hwcws.cahwnet.gov/Bulletins%20General/year_inreview.htm, 1999.
- [DeK99] J. DeKeles, "Terror in the Land of Tech Support," *ZDNet AnchorDesk*, http://www.zdnet.com/anchordesk/story/story_3893.html, Sept. 1999.
- [Etc98] J. Etchison, "Paradigm Schmaradigm," *IT Support News*, Viewpoint, http://www.itsupportnews.com/archives/9807_html/9807view.htm, July 1998.
- [Fre96] A. Freier, P. Karlton, and P. Kocher, "The SSL Protocol, Version 3.0," *Internet Draft Memo*, <http://home.netscape.com/eng/ssl3/draft302.txt>, Nov. 1996.
- [Gia99] G. Gianforte, "Eight Secrets for Successful E-Service," *Supportindustry.com Newsletter*, <http://www.supportindustry.com/newsletter/110299.htm>, Nov. 1999.
- [Gil00] K. Gilhooly, "Certifying the help desk," *IT Support News*, <http://www.itsupportnews.com/july2000/depts/gn/topstory2.htm>, July 2000.
- [Hon00] Honeywell, *Help Desk Services*, http://www.iac.honeywell.com/services/its/call_mgmt/help_desk.htm, 2000.
- [IHS00] IHS Helpdesk Service, *Service Level and Metric Statistics*, <http://www.ihs-helpdesk.com/>, June 2000.
- [ITS99] Editorial staff, "Problem Resolution at Work," *IT Support News*, http://www.itsupportnews.com/archives/9906_html/9906probres.htm, June 1999.
- [Lan00] M. Lane, "Why does knowledge management matter?" *IT Support News*, Viewpoint, <http://www.itsupportnews.com/feb2000/depts/si/sistory1.htm>, Feb. 2000.
- [Loc00] Lockheed Martin Services, Inc., *Help Desk Solutions Center*, <http://www.lmsi-nw.com/it/helpdesk.html>, 2000.
- [Mic00] Microsoft Product Support Services, *Microsoft Knowledge Base*, <http://search.support.microsoft.com/kb/>, June 2000.
- [NCS98] NCSA HTTPd Development Team, *The Common Gateway Interface*, <http://hoohoo.ncsa.uiuc.edu/cgi/>, Jan. 1998.
- [Rea99] B. Read, "Outsourcing a Variety of Support Functions," *Call Center*, <http://www.callcentermagazine.com/article/CCM20000503S0007>, July 1999.
- [RSA00] RSA Security, *Call Handling and Escalation Process*, <http://www.rsasecurity.com/support/techsup/escproc.html>, May 2000.
- [Sch00] Schlumberger, *GeoQuest Customer Support*, <http://www.geoquest.com/pub/support/RGS/> Houston, 2000.
- [Sla00] D. Slater, "Call Center Management," *CIO Magazine*, http://www.cio.com/archive/040100_numbers.html, Apr. 2000.
- [Smi97] G. Smith, "Support for all," *ComputerScope*, http://www.techcentral.ie/cgi-bin/SiteWrapper.pl?template=/magazines/ComputerScope/article_template.html&target=/magazines/ComputerScope/1997/Mar/Features-0.html, Mar. 1997.
- [Ste99] T. Steinert-Threlkeld, "Chatterbot: New answers from customer service," *Internet Business*, <http://www.zdnet.com/zdnn/stories/news/0,4586,2188774,00.html>, Jan. 1999.
- [Sum98] J. Summa, "How to Build a Knowledgebase You Can Use!" *Customer Support Management*, <http://www.customersupportmgmt.com/back/nov-dec98/know.html>, Dec. 1998.
- [Sup00] "E.SURVEY," *Supportindustry.com newsletter*, <http://www.supportindustry.com/newsletter/032800.htm>, Mar. 2000.
- [Ter00] Teradyne, Product: VICTORY Boundary-Scan Software, <http://www.teradyne.com/prods/cbt/products/pVICT/pVICT.html>, 2000.
- [Tol92] L. Tolan, "Managing the High Cost of Distributed Computing," Simon Fraser University Computing Services, http://www.sfu.ca/~lionel/Manage_Cost.html, Dec. 1992.
- [Vir97] Virtual Music Entertainment, Inc, Products, <http://www.virtualmusic.com/products/>, 1997.
- [Wal00] L. Wall, T. Christiansen, and J. Orwant, *Programming Perl, 3rd Edition*, O'Reilly, <http://www.oreilly.com/catalog/ppperl3/>, July 2000.
- [Yah00] Yahoo, Yahoo! search engine, <http://www.yahoo.com>.

FTP Mirror Tracker: A Few Steps towards URN

Alexei Novikov – Institute of Theoretical and Experimental Physics, Moscow, Russia
Martin Hamilton – Loughborough University, UK

ABSTRACT

FTP Mirror Tracker¹ is a software package (written in Perl and C++) that enables transparent, user-controlled redirection to the nearest anonymous FTP mirror sites that are exact replicas of the original source. This redirection can be achieved by using a Web Cache server or by making HTTP requests to the FTP Mirror Tracker directly. The Mirror Tracker also has internal URN support and can be used as a URN resolver for FTP requests. Underlying the system is a MySQL database recording FTP mirror site details. In this report we explain how this database is constructed, and show how it may be used – directly by end users, and under the policy based control of Web Cache and mirror service administrators.

Introduction

Although FTP traffic passing through the modern Internet only accounts for a small fraction of request transactions, its bandwidth utilization is significant. For example, FTP accounted for between 7% and 11% of the incoming traffic on the JANET² network's links to the United States for every month in 1999.

There is a long standing Internet convention that sites which are particularly large (e.g., operating system distributions) or popular (e.g., the Starr Report) will be widely replicated – usually by volunteer effort. The replication process, which typically takes place on a daily basis, is usually referred to as mirroring. Mirroring software exists for replicating Web (HTTP), FTP and (by prior arrangement) arbitrary content, e.g., GNU wget [2], mirror [3] and rsync [4]. In recent years some formalization of this role has taken place, e.g., with the establishment of the UK Mirror Service [5] for JANET users, and the AARNet2 Mirror Archive [6] for Australian academic and research users. Localizing what might well be international (and chargeable) traffic to geographically and/or topologically nearby mirror sites is a challenging task.

A number of approaches to brute-force indexing of the FTP namespace have been attempted, notably the Archie [7] system from Bunyip, CNET's Shareware.com [8] and Lycos/FAST FTP Search [9]. The user interfaces for searching these systems typically allow the end user to supply full or partial filename details or regular expressions to match, and return a list of the URNs where files matching the search criteria can be found. Because of the difficulty in knowing where on the Internet is the best topological/policy match for a client, only minimal attempts have been made to provide tailored output on a per-user basis.

¹<http://squid.itep.ru/mirrored> at <http://www.cache.ja.net/mirrors/MirrorTracker/>

²JANET [1] is the UK's Higher Education and Research Network

The FTP Mirror Tracker Design in Brief

The core of the FTP Mirror Tracker software is a robot which traverses through a list of anonymous FTP servers which it has been told to visit, connects to each of them in turn, fetches their public directory tree content (using the FTP `ls -lR` command), analyses it and creates a unique identifier (actually an MD5 [10] digest value) for the content of each directory and the tree of the directories (using a summarizer program). These identifiers and the FTP URL paths they refer to are stored in a MySQL database [11] using the Perl [12] DBI database interface [13]. In turn, the MySQL database is accessed by a variety of programs which collectively form the “user interface” to the system.

The processed data created by the Mirror Tracker summariser is also made available for use by other services, e.g., for sharing with other Mirror Tracker servers. By default each Mirror Tracker maintains copies not only of the data for the Internet domains it is responsible for, but also the data for the domains which are indexed by other Mirror Trackers. A “root” Mirror Tracker has been established in Moscow, Russia to help bootstrap this process.

From a user standpoint, the basic operation of the FTP Mirror Tracker is to take a URL and return a list of the alternative URLs where this same material may be found – subject to search criteria such as the top level domain(s) required in the search results. This is done by finding the unique identifier associated with the requested URLs, and checking to see whether any other entries in the table(s) in question have the same unique identifier.

We will use the phrase “unique identifier” rather than “MD5 digest value” throughout this paper, since there is no actual requirement that MD5 digests be used as the resource collection identifier. The uniqueness and location independence of our identifiers also makes them attractive as a global naming system, though it should be noted that the way they are

calculated means that they are not persistent over time. This means that they are not truly suitable for use as URNs. However, since there is very little practical deployment experience with URNs, we have chosen to ignore this problem for the time being.

The Gory Details

The FTP Mirror Tracker architecture consists of the following components:

- A robot which collects directory listing data from the FTP servers it has been configured to track.
- A summarizer which creates MD5 digests of the directory listings.
- A digest exchanger, for sharing the digests with other servers.
- A back end database – currently MySQL.
- Various frontend programs.

We will describe in short the design of each of these components, and how they have been implemented.

The Mirror Tracker robot

The function of this component is to gather raw directory listings from FTP servers for processing by the summarizer. The robot has been implemented as two Perl programs – robot, which parses the list of the FTP servers for each domain begin indexed and forks a second Perl script, ftp_list, to actually fetch the directory listing for the server.

If it was able to start an anonymous FTP session with the target server, ftp_list proceeds to check whether there is an ls-lR.gz file in the root or /pub directories on the server. If one that it is reasonably fresh (and not empty) is present, this is fetched, otherwise the ls -lR command is issued from the top level directory of the anonymous FTP server in order to produce a recursive directory listing.

The summarizer

This is the heart of the FTP Mirror Tracker. Its job is to parse the directory listings of the FTP servers which are being tracked (fetched by the robot), analyze them, and create MD5 digests based on this analysis. The summarizer is implemented by a C++ program, createdigest, which reads in the raw directory listings and generates a list of MD5 digest values and URLs on a per-directory basis.

It might seem sensible to use all of the information from the directory listing output when creating the digest of it (i.e., permission, node, owner, group, size, date and the filename). Unfortunately most of these values can be changed during the mirroring process. In practice it seems that we can get by using only the file size and name (though as noted, some mirrors will compress files – we exclude these), plus the file type (e.g., plain file, directory or link).

We will note in passing that a better approach to persistence would be to calculate unique identifiers based on the contents of the files themselves, rather than the directory listing metadata. A program has been provided with the FTP Mirror Tracker distribution to let FTP server administrators calculate MD5 digest values for each file in a given directory hierarchy. However, this is very much a “future”, since it would have to be widely deployed in order to be generally useful.

```
total 821
-r-xr-xr-x 1 root ftp 62163 Jan 25 19:43 compress
-r-xr-xr-x 1 root ftp 168240 Jan 25 19:38 date
-r-xr-xr-x 1 root ftp 106752 Jan 25 19:38 gzip
-r-xr-xr-x 1 root ftp 186848 Jan 25 19:37 ls
-r-xr-xr-x 1 root ftp 270232 Jan 25 19:38 tar
```

Listing 1: Typical output of the ls -lR command.

So, in order to produce the digest of a directory, we take the size and name of each of the files within it and concatenate them, e.g., in the example above we are left with:

```
62163compress168240date106752gzip186848ls270232tar
```

We then create a hexadecimal representation of the MD5 digest value for this string and assign it to the URL of the original directory on the FTP server. In this way we can create a relatively small unique identifier for the directory contents rather than for the files themselves. Finally, we add this text to the parent directory listing string, giving an identifier for the contents of the complete directory structure on the FTP server.

To understand why we need to add the unique identifier of the subdirectories to the string representation of the parent directory, let's take a simple example. Let's assume that someone wants to fetch the latest XFree86 for Linux running on x86 with glibc 2.1 from the nearest server. This person starts to look for an exact replica nearby; see Table 1. If we hadn't

1	ftp://ftp.xfree86.org	no exact replicas
2	ftp://ftp.xfree86.org/pub/	no exact replicas
3	ftp://ftp.xfree86.org/pub/XFree86/	still no exact replicas
4	ftp://ftp.xfree86.org/pub/XFree86/4.0.1/	found some! so he shifts to the nearest one
5	ftp://ftp.gamma.ru/3/XFree86/4.0.1/	
6	ftp://ftp.gamma.ru/3/XFree86/4.0.1/binaries/	
7	ftp://ftp.gamma.ru/3/XFree86/4.0.1/binaries/Linux-ix86-glibc21/	

Table 1: Search sequence for exactly replica.

added information about the subdirectories to the directory information, we couldn't be sure that the Linux-ix86-glibc21 directory is located somewhere on the mirror server under the 4.0.1 directory.

The Database Back End

We use a simple database design, with a dedicated database for the FTP Mirror Tracker data, and separate tables within this for each of the digest and link collections for each of the top level domains being tracked.

We are able to take advantage of the MySQL "load data" feature to read in the digest and link files directly, with MySQL automatically creating a database row in the appropriate table for each line of these files. The actual database manipulation is done using Perl, with DBI for database access.

The Digest Exchange

The original files are compressed and are moved to a directory which is accessible through the WWW, so that they can be shared with other Mirror Trackers.

Using the method described above we parse, for example, 100 MB of compressed listings from FTP servers (representing almost all of the anonymous FTP

servers in Germany), create a 50 MB digest file and a 25 MB links file for feeding into the database engine. After compression, these files are 7 Mb and 2 Mb respectively – small enough to be shared easily via the WWW.

Frontend Programs

We provide a simple CGI program (written in Perl and C) which lets the end user query the FTP Mirror Tracker for specific URLs. This can be interacted with directly by the user, or linked to by content providers. There is also a third mode of operation which we will call Mirror Tracker on Demand.

Mirror Tracker on Demand is a JavaScript command which can be saved as a bookmark or (for example) placed on the Netscape Communicator Personal Toolbar (shown wrapped):

```
javascript:location.href='http://
tracker.foo.bar/cgi-bin/
tracker?url=' +
escape(window.location);
```

Pressing this button will cause the current URL to be sent for resolution by the FTP Mirror Tracker, and the results of the Mirror Tracker search to be

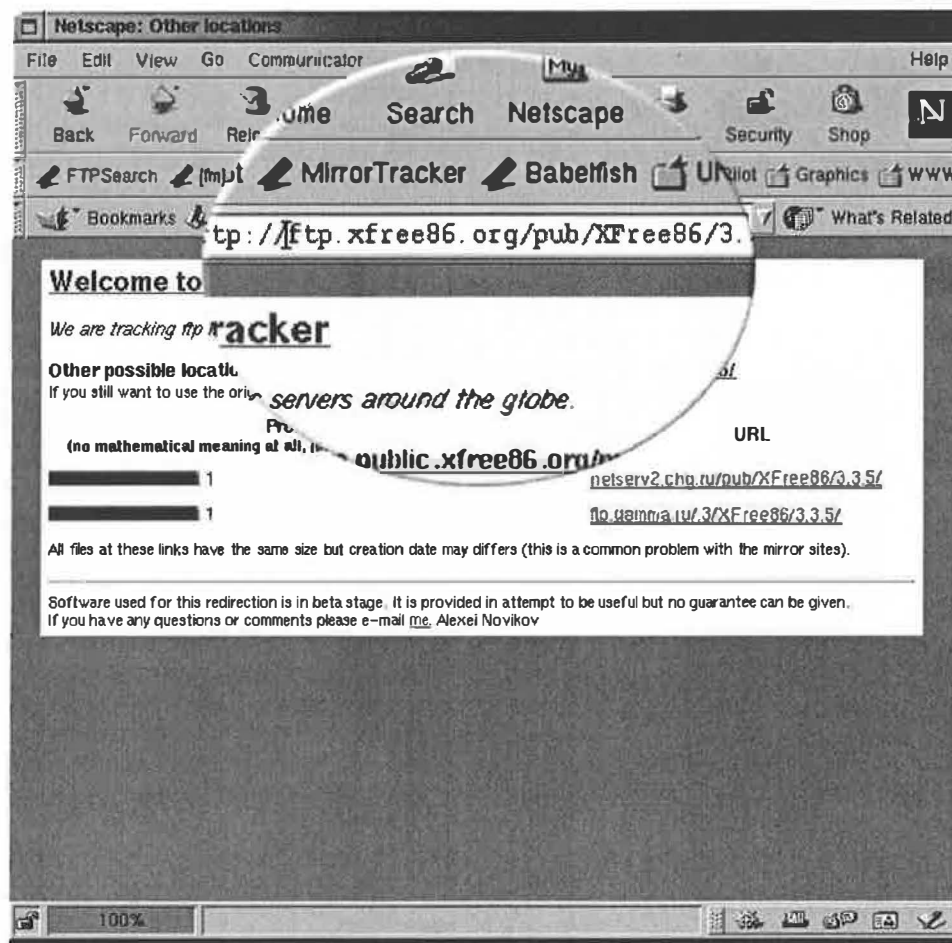


Figure 1: Typical browser window with the output from the WWW frontend. "Mirror Tracker" button is zoomed.

presented in the browser's main window – as shown in Figure 1.

Although these user interfaces are very simple, they are also extremely powerful – instead of using search engines periodically to find out who is mirroring which sites, and trying to figure out which mirror sites are fresh and which are stale, content providers can have just one link on their site to the nearest FTP Mirror Tracker. The Mirror Tracker software will provide users with the list of the sites which have fresh and accurate replicas, optionally located in the same domain as the user.

Content providers only have to register with their nearest FTP Mirror Tracker server and put a link to it on their Web page. The link will look something like this (show here with lines wrapped for display purposes);

```
<a href="http://squid.itep.ru/
  cgi-bin/tracker?url=YOUR URL">
  YOUR PACKAGE</a>
```

For example, the following link would yield all of the mirror sites known for the XFree86 3.3.6 release (shown wrapped):

```
<a href="http://squid.itep.ru/
  cgi-bin/tracker?url=ftp://
  ftp.xfree86.org/pub/XFree86/
  3.3.6/">3.3.6 Version Mirrors</a>
```

Web Cache and Mirror Service Integration

Architecture for Cache/Mirror Cooperation

Whilst there is no standard network protocol for building and interacting with mirror services (rsync is probably the closest thing we have to this), there are a variety of ways in which Web Caching systems can be made to interoperate with each other – and in turn with Mirror services. In addition to the core proxy HTTP service, most Web Cache products also support interoperability via the Internet Cache Protocol [14], and the freeware Squid Web Cache [15] supports the new Cache Digest protocol [16]. Other cache cooperation protocols exist, such as the HyperText Caching Protocol [17], but these have yet to be as widely deployed and will not be further considered in this paper.

Since we do not wish to build our own Web Cache server, we have opted to use Squid for proxy HTTP services and graft on our own ICP server and Cache Digest generator – these are described in greater detail below.

Many Web Cache packages provide a URL rewriting facility, although some of these are limited to taking in a list of URLs to rewrite and the URLs to rewrite them to – or a list of URLs to block access to. So we can redirect desired URLs to the server with the FTP Mirror Tracker resolver.

Squid's approach, the “redirector” program, is particularly well suited to use with the FTP Mirror Tracker, since (when it is enabled) URLs are passed to

an external program for analysis and potential rewriting. For performance reasons (the external program doing this may block whilst waiting for an operation to complete) Squid forks a large number of redirector programs which run continuously in parallel to the main Squid server process.

Use of the Internet Cache Protocol (ICP) for Cache Cooperation

The Internet Cache Protocol allows a Web Cache (or any other program which is interested in talking to Web Caches) to query another Web Cache as to its contents. Each ICP query or response takes the form of a UDP packet (with all fields in “network” or “big-endian” order), consisting of a twenty byte header with fields and a variable length payload. Typically the payload consists simply of a URL.

ICP is, for all its problems, the most widely implemented Web Cache cooperation protocol, available in most freeware and commercial caching products. Consequently, we decided to support it as an access method for the FTP Mirror Tracker too. Rather than modify Squid's own internal ICP server, which is very closely tied to the rest of Squid, a Perl module WebCache::ICP [18] was written to encapsulate the details of ICP packet processing within a simple object-oriented interface. This gives us an easy way to provide an ICP service which returns whatever ICP response we like on a given ICP request.

The ICP server we use with the FTP Mirror Tracker queries of the Mirror Tracker database to determine whether the Mirror Tracker should be visited for a given URL in the ICP request packet. In the case of downstream caches which are running Squid, we can reduce the amount of ICP traffic associated with the peering by using the “cache_peer_access” Access Control List, e.g., to specify that only FTP URLs should be sent to this peer.

Use of Cache Digests for Cache Cooperation

Cache Digests (implemented in Squid since version 1.2beta) offer a completely different approach to sharing information about cached objects, but have yet to be widely deployed in commercial Web Cache products. We decided to include them in the FTP Mirror Tracker project because of their widespread use with Squid, but the reader should note that they are still classified as experimental and subject to change.

Whereas ICP requires request and response UDP packets to be exchanged, Cache Digests work on the principle that the Web Cache builds a summary of its contents. This summary takes the form of a 128 byte header, followed by a bit array. This summary is then made available to other Web Caches via the normal proxy HTTP interface, and these are then in a position to do a local (Digest) lookup when trying to determine whether any of their peers has a requested URL.

In creating a Cache Digest populated from the Mirror Tracker database, we have the problem that there is no way to register whole URLs – the Mirror

Tracker database currently does not contain this information. Fuller Cache Digests could, however, be constructed using the raw directory listing information which is processed in order to create the Mirror Tracker database.

Uniform Resource Names (URNs)

Introduction to URNs

URNs [19] are being developed by the Internet Engineering Task Force as an alternative naming scheme, complementary to the existing URL. Whereas URLs essentially encode a "recipe" of instructions to be followed when fetching a given copy of a resource, URNs are required to be:

- Location independent, so that the instructions for reaching the resource are de-coupled from the name by which the resource is cited, e.g., in Web documents.
- Persistent, so that the assignment of a URN to a resource effectively acts as a guarantee that the resource will continue to be accessible by this name indefinitely and will never be replaced by another resource which has the same URN.
- Compatible with existing name spaces such as ISSN and ISBN numbers.

In practice this is intended to be done by defining a number of "namespaces", each of which will have its own procedures for things like registration and management. URNs themselves [20] are simply the concatenation of the string "urn", a centrally assigned (by the Internet Corporation for Assigned Names and Numbers [21]) identifier for the namespace in question, and then a "Namespace Specific String" – separated by colons. That is:

urn:Namespace Identifier:Namespace Specific String

The authority responsible for each namespace [22] is free to subdivide the Namespace Specific String component of their URNs as they see fit – subject to character set encoding requirements which are designed to ensure that URNs may be used on the widest possible range of devices. The Namespace Identifier itself is chosen out of the set of upper and lower case letters, numbers and the hyphen character -, though the first character may not be a hyphen. The Namespace Specific String has a slightly larger vocabulary, which includes some punctuation characters.

URNs and the FTP Mirror Tracker

As noted above, the Mirror Tracker unique identifiers are subject to change over time. This means that they cannot be true URNs as these are defined by the IETF. However, since there is still a dearth of true URNs for people to experiment with, we shall ignore this limitation for the moment.

We supply a Squid compatible "N2L" [23] program with the FTP Mirror Tracker, using the Mirror Tracker "unique identifiers" as URNs and the experimental x-tracker Namespace Identifier. An example of a Mirror Tracker URN would be:

```
urn:x-tracker:57ce083433061aab97c9c2b63759ef2f
```

This is the unique identifier for all of the copies of the directory listing which can also be referred to by the URLs (as in the summarizer examples above):

```
ftp://ftp.xfree86.org/pub/XFree86/4.0/
ftp://netserv2.chg.ru/pub/XFree86/4.0/
ftp://ftp.gamma.ru/.3/XFree86/4.0/
ftp://caramba.cs.tu-berlin.de/pub/X/XFree86/4.0/
ftp://wizard.freeware.com/.0/XFree86/4.0/
ftp://ftp.linux.tucows.com/pub/XFree86/4.0/
```

Unfortunately, since even Netscape must be explicitly configured to talk to a proxy for URN resolution, we cannot simply deploy support for URNs as part of (for example) the JANET Web Cache Service and have them immediately be available to the end user. Each of the sites which connects to the service would need to enable URN support in their own users' browsers too.

Summary

The FTP Mirror Tracker enables transparent, user-controlled redirection to the nearest FTP mirror sites which are exact replicas of the original source. The redirection can be achieved by using a Web Cache server or by making an HTTP request to the FTP Mirror Tracker directly. The Mirror Tracker has internal URN support and can be used as a URN resolver for FTP requests.

During the course of this work we have produced a variety of tools and documents which may be useful for other purposes above and beyond the FTP Mirror Tracker itself, e.g., the ICP and Cache Digest Perl modules, the Cache Digests specification, and the FTP robot. The software, databases, and documentation associated with the project are available for download via its homepage <http://squid.itop.ru/>. The changes which we made to Squid have been folded into the mainstream Squid 2.3 distribution, released in January 2000.

Acknowledgments

Discussions with Dave Beckett, Rodrigo Castro, Iain Fothergill, Claire Moore, George Neisser, Mark Russell and Michael Sparks are gratefully acknowledged.

The authors are grateful to the Trans-European Research and Education Networking Association for financial support under the Pilot Project initiative, and the Institute of Theoretical and Experimental Physics for hosting the FTP Mirror Tracker primary server and providing bandwidth for its operations.

Author Information

Martin Hamilton is a member of the Department of Computer Science, Loughborough University, UK & JANET Web Cache Service. Reach him electronically at martin@wwwcache.ja.net.

Since August 1997 Martin has been working on the JANET Web Cache Service, in the Computing

Services department of Loughborough University. The JANET Web Cache Service is funded centrally (with a contract awarded by open tender) for the use of the UK Education and Research community (users of the JANET network). With the instigation of usage-based charging for traffic from the US to JANET, this service has become extremely popular and has accounted for as much as half (46% on January 30th 2000) of the Web traffic transferred to JANET from the US. The JANET Web Cache Service is built upon open source software – the Linux and FreeBSD operating systems and the Squid Web Cache server.

In his spare time, Martin works as a volunteer on the GNU free software project at Massachusetts Institute of Technology and has contributed code to several popular open source products – including the NCSA and Apache World-Wide Web servers, the NCSA Mosaic WWW browser, and the Squid Web Cache.

Alexei Novikov has been a researcher at the Institute of Theoretical and Experimental Physics, Moscow, Russia. His email address is anovikov@heron.itep.ru since 1998. He defended his Ph.D. thesis (*Theoretical Restrictions on the Possible Extensions of the Standard Model Based on LEP Data*) in 1998. He works in the field of theoretical High Energy Physics (six papers in refereed journals, seven talks at the international conferences). He is interested in computer science and is developing several open source projects.

Bibliography

- [1] *JANET website*, <http://www.cache.ja.net/>.
- [2] *GNU wget homepage*, <http://www.gnu.org/software/wget/>.
- [3] *Mirror homepage*, <http://sunsite.org.uk/packages/mirror/>.
- [4] *rsync homepage*, <http://rsync.samba.org/>.
- [5] *UK Mirror Service*, <http://www.mirror.ac.uk/>.
- [6] *AARNet2 Mirror Archive*, <http://www.aarnet.edu.au/projects/>.
- [7] *Archie website*, <http://archie.emnet.co.uk/>.
- [8] *CNET shareware.com website*, <http://shareware.cnet.com/>.
- [9] *Fast FTP Search website*, FAST ASA, <http://ftpsearch.lycos.com/>.
- [10] R. Rivest, *RFC 1321, the MD5 Message-Digest Algorithm*, URN:ietf:rfc:1321, April 1992.
- [11] *MySQL website*, <http://www.mysql.com/>.
- [12] *Perl website*, <http://www.perl.org/>.
- [13] *Perl DBI modules* at the Comprehensive Perl Archive Network, <http://www.cpan.org/modules/by-module/DBI/>.
- [14] Wessels, D. & K. Claffy *RFC 2186, Internet Cache Protocol (ICP), version 2*, URN:ietf:rfc:2186, September 1997.
- [15] *Squid Web Proxy Cache Website*, <http://www.squid-cache.org/>.
- [16] Hamilton, M., A. Rousskov & D. Wessels, *Cache Digest Specification – Version 5*, <http://www.squid-cache.org/CacheDigest/cache-digest-v5.txt>, December 1998.
- [17] Vixie, P., & D. Wessels, *HyperText Caching Protocol*, URN:ietf:rfc:2756, January 2000.
- [18] M. Hamilton, *WebCache::ICP Perl Module* at CPAN, <http://www.cpan.org/modules/by-module/WebCache/>.
- [19] K. Sollins, *RFC 2276, Architectural Principles of Uniform Resource Name Resolution*, URN:ietf:rfc:2276, January 1998.
- [20] Moats, R., *RFC 2141, URN Syntax*, URN:ietf:rfc:2141 May 1997.
- [21] *Internet Corporation for Assigned Names & Numbers website*, <http://www.icann.org/>.
- [22] Daigle, L., D. van Gulik, R. Iannella, P. Falstrom, *RFC 2611, URN Namespace Definition Mechanisms*, URN:ietf:rfc:2611, June 1999.
- [23] Daniel, R., *RFC 2169, A Trivial Convention for using HTTP in URN Resolution*, URN:ietf:rfc:2169, June 1997.

Deployme: Tellme's Package Management and Deployment System

Kyle Oppenheim & Patrick McCormick – Tellme Networks

ABSTRACT

Many administrators use a central software repository because managing distributed packages is difficult. Deployme is our solution to manage the package update lifecycle across a large number of independently configured hosts.

Deployme is highly flexible and has been extended to handle many different types of packages. Deployme packages include standard UNIX tools, local applications, web site content, and voice site content. Most packages require fast, frequent deployment.

Deployme has a web-based user interface that allows less technical users to deploy on their own. Deployme also restarts appropriate server processes, a feature which was much more difficult than we expected.

We discuss other lessons learned from the first implementation and planned improvements for the future.

Introduction

Using small, independent software packages is a common approach to managing software. Many tools have been developed to address package management including Depot [Manheimer1990] and LUDE [Dagenais1993].

The package philosophy behind these tools is easily extended to rapidly changing content. However, we found these systems to be incomplete because they only cover one or two stages of the package update lifecycle.

Similar to the software distribution cycle outlined in [Furlani1996], we have divided the package update lifecycle into four steps:

1. Create the package.
2. Distribute the package to designated hosts.
3. Install and activate the package on each remote host.
4. Delete old, unused packages from remote hosts.

Deployme automates package creation and eliminates the need for a release engineer to manually package data. (As you may have guessed, "Deployme" is one of many tools derived from Tellme's corporate moniker.) Deployme exports source data from CVS [Cederqvist1993] and then builds executables and other generated files.

The system also handles installation, activation, and removal of packages, completing the package lifecycle. Deployme automates most of the traditional release engineer role.

The original design was intended to manage and deploy web site and voice site content. We have since found the system to be general enough to be used for local applications, common tool management, and other types of content.

With Deployme, our producers and developers manage their own releases. A straightforward web user interface allows the development team to update the Tellme service without the intervention of our network operations team.

Motivation

Our web-based content management system before Deployme was difficult to maintain, but was easier to use than we expected. This success led us to make a simple user interface a major Deployme goal.

As applications passed quality assurance, content developers checked their changes into a CVS repository. Once all applications had reached a known good state, the tree could be released.

To perform a release, a developer would go to a web page with a single button on it. The button started a shell script that checked out the CVS content tree and copied the data to the production server.

Platform releases were similar. Again, we built a web page with a single button that kicked off a shell script. The script performed a CVS pull, ran the build, and copied the release to the production server. As more servers arrived, we added them to the list that received completed builds.

The benefit of the release scripts was that they took much of the manual work out of releases and freed up developers to do more constructive tasks. The scripts also significantly reduced developer stress, as platform releases became an everyday occurrence instead of a frightening event. Finally, they allowed any developer to be appointed release engineer, because the only process involved was filling out the form on the web page.

The script proved increasingly difficult to modify as we increased the number of build targets and

machines. It also could not support features such as rollback. If we wanted to push the same build to a new machine using the script, we had to wait while the script recompiled the platform from scratch. It became obvious that our inefficient distribution script could not scale.

At this point, we seriously considered having our production servers attach to a storage area network (SAN) instead of distributing software packages to local storage. We decided against this for several reasons.

Distributing packages to local disks still provides high availability, but it is cheaper and more flexible. A centralized file repository does not span across cities and continents without significant cost in time, hardware, and complexity. Also, most centralized solutions would lock us to one vendor. Our failures are restricted to individual machines, not an entire storage array. Even with high-availability components, SANs still act as a single point of failure.

So, to replace the unscalable release script, we wrote a new script to perform releases. This script was the first implementation of our distributed package architecture. However, it only performed distribution and activation of packages. It had no web interface, requiring a network operations engineer to serve as gatekeeper for all releases of content, platform, tools, and configuration files.

Updates to the production system became very infrequent. Developers plaintively asked, "Why can't we bring back the button?"

Goals

Deployme's mission is to provide a central system for tracking the entire lifecycle of software packages. We established the following goals to judge the success of the project:

- *Support a wide audience.* Users who are less inclined to system administration tasks should be able to create and deploy packages. A human gatekeeper will impede updates.
- *Robustness.* With a wide audience using the system, the system must never activate a package without verifying that it was distributed successfully. Detailed audit logs and notification must be included to inform our Network Operations team of system changes.
- *Augment the development process.* Packaging should not be a complex task, but instead it should be seamlessly integrated into existing processes.
- *Flexible destinations.* Packages should be able to be deployed to various destinations such as development systems, testing systems, and production systems.
- *Efficient use of network bandwidth.* When large groups of servers are located at remote data centers, Deployme should only push bits over

the slow link once and distribute where there is good bandwidth.

- *Quick pushes.* The system should allow, but not enforce, staging of a package for testing. If a "hot fix" (e.g., a critical bug fix or a press release for the web site) must be pushed immediately, it should be possible to bypass staging servers.
- *Seamless activation.* End users (customers) should never be aware of a new package being pushed (e.g., Web site links should never be broken). If a server binary is being pushed, the server process should be gracefully restarted.
- *Rollback.* It should be possible to return to a known good state quickly and robustly.
- *Scalability.* Deployme needs to be able to handle hundreds of modules, hundreds of servers, and thousands of packages.

Non-Goals

We intentionally chose not to address several issues in the first implementation.

- *No local package management.* Enough robust single-server package managers have been developed in the past that we should not invent a new one.
- *No dependencies.* Our aggressive schedule would not leave us enough time to develop dependency tracking properly. We felt we could achieve all of the goals above without dependency tracking.
- *No fine-grained operations control.* While we chose to integrate certain operations functionality into Deployme, we decided against turning Deployme into a "control panel" for controlling remote processes.

Previous Work

There is much previously published work on package management and deployment tools. Many tools, such as Depot [Manheimer1990], take advantage of transparent remote network file system access such as NFS [Sandberg1985] or AFS [Howard1988]. A system that contains a centralized database containing server configuration information is presented in [Finke1997]. Microsoft has an architectural discussion of their corporate website publishing tool in [Moore1999].

Various methods for remote execution and server process restarting have been presented in tools such as Igor [Pierce1996] and Synctree [Lockard1998].

As stated in the previous section, Deployme does not address local package management directly. Instead, it calls routines to "activate" and "deactivate" packages on a machine. In our environment, we use GNU Stow [Glickstein1996] for this purpose.

Design and Implementation

The biggest problem with the old content release process was that all content was released at once.

Every tree push was potentially catastrophic. If any one application had some code checked in that had not been fully tested, we would have to roll back the entire tree. We learned that a big tree of content is more difficult to test than a big tree of code.

We decided to rearrange the entire content system in order to make it possible to release one application at a time using Deployme.

Deployme is written entirely in Perl5 [Wall2000]. We chose Perl because of our familiarity with it, its straightforward database integration, and our desire to attempt a complex, structured project in the Perl language.

Deployme has a simple three-tier architecture: the user interface, the rules logic, and the database.

User Interface

The Deployme interface is a set of web pages that display a list of available products and a list of potential target servers. There are also information pages that show which packages are active on a given server.

Each product is called a “module”, after the CVS term. A product could be a collection of web pages, or the source required to build an executable.

Each release of a module is called a “tag”, again after the CVS term. A tag corresponds to a physical package on disk. The package is assembled by requesting that the version control system write out the tagged version of the module.

The central assumption of Deployme is that the contents of a tag never change. This is stricter than the version control sense of “tag”, since most version control systems allow users to move a tag forward or backward on a given file's timeline.

Because of this assumption, Deployme's tag namespace is a superset of the version control tag namespace. For example, the CVS tag “engine-1” can be tagged in Deployme as “engine-1.DEBUG.i386.solaris.5_6” or “engine-1.RELEASE.sparc.solaris.5_7”. The source code is the same, but the resulting package is compiled with the appropriate build flags and architecture.

By assuming that packages never change, we can skip crosschecking between servers to establish the version of a package. This also means that the slightest modification requires a new tag.

To start a Deployme job, the user first picks a set of tags. The user is restricted to only selecting one tag for each module, because Deployme does not allow two versions of the same module to be active at the same time. Then, the user picks a set of servers. Deployme checks to ensure that the request does not break any rules, and then executes a job fulfilling the request.

To rollback a module, the user selects a previous tag and follows the same procedure.

Rules Logic

The middle tier comprises the rules logic that creates and executes a deployment job.

[Click for recent jobs <](#)

Module Deployment:

Select a view (currently viewing: vui):

[vui Developer](#)
[Platform Developer](#)
[All Modules](#)

[www Developer](#)
[Netops](#)

Type	Contact	Module	Release	Select <input type="checkbox"/>
vui	jeremy	airlines	<input type="text" value="vui-airlines-20000908-1743.RELEASE"/>	<input type="checkbox"/>
vui	eddy	blackjack	<input type="text" value="vui-blackjack-20000804-1302.RELEASE"/>	<input type="checkbox"/>
vui	eddy	horoscopes	<input type="text" value="vui-horoscopes-20000906-1949.RELEASE"/>	<input type="checkbox"/>
vui	eddy	news	<input type="text" value="vui-news-20000910-2155.RELEASE"/>	<input type="checkbox"/>

Servers:

QA Content ☐

☐ content.qa.tellme.com

Production Content ☐

☐ content01.tellme.com

☐ content02.tellme.com

Do NOT LINK modules ☐

Do NOT RESTART modules ☐

Figure 1: The Deployme web interface.

Deployme's logic falls into two categories: job creation logic and job execution logic. The former uses a set of rules to determine what tasks should be done. The latter uses the database state tables to carry out those tasks.

Job creation expands the list of packages and destinations into a series of tasks. This step is triggered when the user submits a request form in the web UI. See Table 1.

In order to figure out which tasks are necessary, the job creation algorithm relies on each module's type; see Table 2.

Module Type	Detail
platform	Source code that needs to be compiled.
www	Website content.
vui	Content for our voice-driven service.
tool	Packages that need no compilation; vendor binaries, tools, and configuration files fall into this category.

Table 2: Deployme module types.

The rules are generally very simple at this stage. For example, platform modules need a BUILD step; the others do not.

For the PUSH step, Deployme first finds out if the package is already on the destination server. If the package is already present, we skip the PUSH step, because we assume that packages never change.

This optimization makes rollback extremely fast. No data needs to be copied to the server; instead, we progress immediately to the LINK step that simply reactivates the old package.

If a PUSH is necessary, the job creation algorithm determines what route the package should take to the destination. In some cases, a single PUSH step is all that is required to copy the package from the Deployme master package repository to the remote location. However, if the final destination is at a

remote facility, Deployme will instead add PUSH1 and PUSH2 steps.

The PUSH1 step sends the package to a machine at the remote facility which is designated as a gatekeeper. This machine serves as a cache of the master repository. The PUSH2 step asks the gatekeeper machine to copy the package from the cache to another machine at the same facility over the high-speed local network.

Since most of our machines are in remote locations connected by low-bandwidth connections, this logic saves hours of file transfer time.

The Database

Using a database on the backend makes it incredibly easy to provide a rich web UI. Also, a database allows operators to create highly refined reports using SQL queries. Deployme requires that the backend support SQL and the Perl Database Interface (DBI). We selected the freely available MySQL RDBMS [MySQL2000] as its cost and speed were attractive. Also, the current version of Deployme does not require any advanced database support such as transactions.

The database schema has tables that represent the visible items in the UI: modules, tags, and servers. There are additional tables, like "servergroups", that aggregate rows for a prettier display. Deployme can skip tasks that have already been done by previous jobs. The "state" table tells us which packages are active on which servers, and the "history" table holds jobs in all stages of execution.

One problem with using a database to record machine state is that the database can become unsynchronized with the real world. For instance, if a given server is replaced or if an operator performs a manual package upgrade, the database will become stale. This issue was not much of a problem for content pushes, but as we extended Deployme to include platform and tool pushes to hundreds of servers, the inconsistencies began to pile up.

We attack this problem through sanity checks and a reconciliation script. The sanity checks are performed both during job creation and execution. For

Lifecycle Stage	Deployme Task	Detail
Create package	PULL	Export package from source control.
	BUILD	Build executable or content from sources or templates.
Distribute package	PUSH	Push package directly to destination host.
	PUSH1	Push package to intermediate host.
	PUSH2	Push package from intermediate host to destination host.
Install and activate package	LINK	
Remove obsolete packages	CLEAN	

Table 1: Deployme tasks.

example, if you attempt to push a SPARC package to an Intel machine, we stop the job before it starts.

The reconciliation script scans the packages on remote machines and compiles a list of active versus inactive packages. This list is compared against the database. The check script then updates the database to match the actual state of the server.

The many problems we encountered with database consistency sparked a debate about whether the database should contain any state information at all. Another approach would be to make all job execution decisions based on the actual state of the server instead of a cached version. However, we decided against local state in this version of Deployme due to the time it takes to inventory a single server and our desire to preserve the database's excellent reporting capability. We will revisit this issue in version 2.

Job Execution

Most of Deployme's code is in the job execution system.

After job creation is complete, all tasks are filed in the database's history table. The web UI then kicks off a background process to execute the new job.

The job execution system is modular. We split Deployme into a set of Perl packages designed to handle different cases for each task.

The first tasks are generally PULL tasks. These tasks use CVS to retrieve the appropriate tag for the given module. Once the tag is pulled into a package, the state table is updated.

BUILD tasks are currently only performed for platform builds; the build task executes "make" and verifies build completion. The build module finds a build server in the server table and then uses ssh [Ylonen1996] to remotely start and monitor the job. All build servers mount the master package repository via NFS and write the build results to the package, thus completing package formation.

PUSH tasks are performed using either the rsync utility [Tridgell2000] over ssh, or the rsync utility with a listening rsync daemon. Configuration information in the database indicates which transport to use. Other transports can be added easily in the future.

For PUSH2 tasks, we open an ssh connection to the intermediate server, and then initiate an rsync over ssh to the final server.

We choose the appropriate code module for the LINK task based on the module type that is being pushed.

Linking "www" and "vui" modules types is relatively simple. Package activation often boils down to maintaining a few symlinks that point to the current packages. For our voice site, we also update the list of available keywords appropriately.

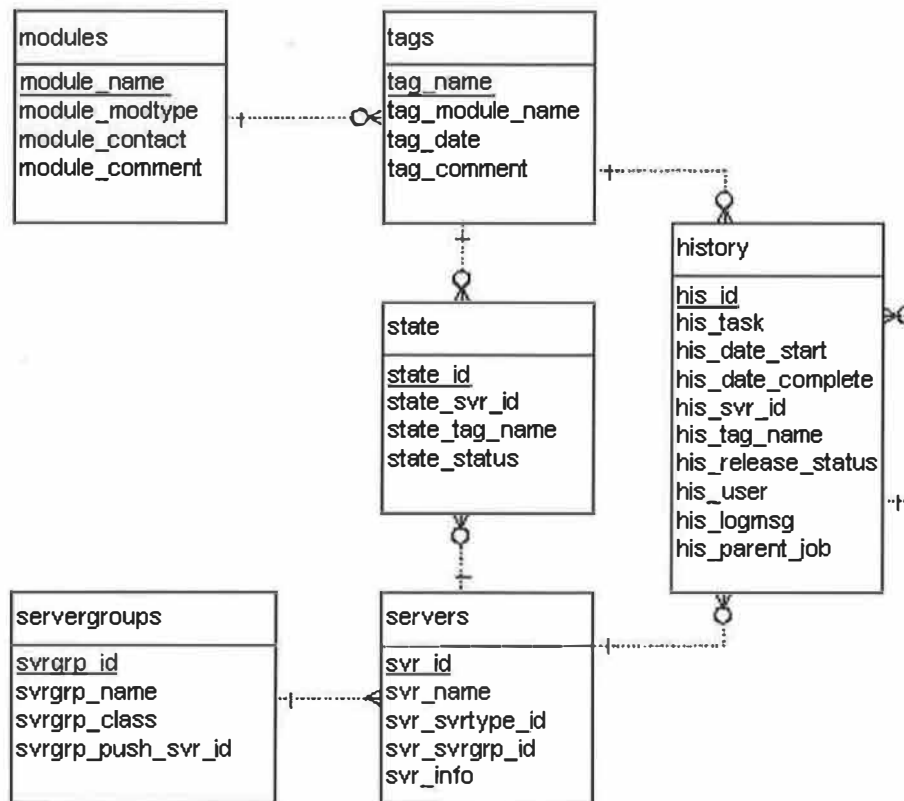


Figure 2: Database schema.

The “platform” and “tool” module types have the most complicated LINK step. We adopt the policy that any software that is running should be restarted when linked. For this reason, the LINK step needs to figure out which processes should be shut down before the link and restarted afterward. In some cases we also need to wait for processes to come back up.

As producers and developers are free to do their own releases, the extent of our release engineering responsibilities boils down to when to run CLEAN jobs. Cleanup has its own job creation logic that determines which packages are eligible for cleanup. For each module we specify how many old packages to keep on local disks and the minimum age of a package before it is eligible for deletion. Keeping some old packages allows us to do fast rollbacks since the package is already at the destination – just a quick re-link is required. Using a minimum age is an attempt to make sure we have a good package to roll back to, and not just a series of bad packages that were released in quick succession. The cleanup logic examines the content of each server and creates a list of packages to delete with the additional constraint that no active packages can be removed. A cleanup job is created that is executed similarly to a deployment job.

The Problem With Restart

The decision to add process restarts to Deployme proved to be much more difficult than we first realized.

We first tried to create some general rules about what kinds of processes to restart on a machine, and only included certain critical processes in the rules. This group of special cases was small and we wrote them into the code base. As Deployme expanded, however, it quickly began to encompass many more programs than we had planned.

This solution became a small dependency tree. This violates one of our Non-Goals discussed above. Because the dependency tree was restricted to platform LINK tasks, and because it remains rather small, we felt this transgression was minor. It turned out to be a continuing headache as we used Deployme for more modules and more servers.

From a conceptual viewpoint, “restart” does not fit into the package lifecycle. We rolled it into the “Link and activation of package” step, which increases the possibilities for failure in the LINK task.

There are significant ordering issues involved with restarting processes. While each task only relates to a given server/tag pair, the LINK step must aggregate all tasks for a given server together and then determine in what order to shut down the dependent processes. Order is determined using a small table in the database (not shown in the schema) that contains ranked module-process dependencies.

Here is an example that shows how this can become complicated. Assume we have a job pushing new telephony firmware along with new monitoring software. The firmware requires that the telephony driver and the telephony server be shut down, where the telephony driver has higher priority. The monitoring software requires that the monitoring process be shut down. For maximum efficiency, the aggregator creates a job that will shut down first the monitoring software, then the telephony server, and then the telephony firmware. All links are done, and then the software is restarted in reverse order. We also add a special-case delay for the telephony firmware since it requires a half-minute to reconfigure after the load is complete.

The ordering problem shows that it is not enough to just implement dependencies between modules and processes. We need to explicitly represent dependencies between processes and other processes.

We were correct to not spend too much effort tackling the dependency problem early on. Our implementation would have likely been incorrect. Having completed the first implementation of Deployme, we now understand what kinds of dependencies should be tracked.

Parallel Remote Execution

Another area of complexity is the method for triggering jobs on remote servers.

The remote part of the platform link stage is too complicated to use individual ssh statements, so instead we created the “linkworker”. The linkworker is a small Perl script with no dependencies that can be used to perform link, unlink, start, and stop operations on a remote host.

Even if a link only takes a minute to perform, doing links serially across many servers takes a very long time. Serial processing severely inhibits

Detail for job #31819 [\[view log\]](#)

Task #	Task	Started	Completed	User	Module	Tag	Server	Info	Status
31820	PULL	2000-08-04 13:08:10	2000-08-04 13:08:10	eddy	blackjack	vui-blackjack-20000804-1302.RELEASE		Packages pulled from CVS.	
31821	PUSH	2000-08-04 13:08:10	2000-08-04 13:08:32	eddy	blackjack	vui-blackjack-20000804-1302.RELEASE	content.tellme.com	Package pushed	
31822	LINK	2000-08-04 13:08:32	2000-08-04 13:08:33	eddy	blackjack	vui-blackjack-20000804-1302.RELEASE	content.tellme.com	Linking package	INPROGRESS

Figure 3: Deployme status.

scalability. To help meet our scalability goal, we created a "parallel_mommy" routine that parallelizes all links.

When a linkworker is started, the mommy passes a "flag file" argument. As the linkworker progresses through the jobs, it records what it has done in a flag file, and dumps logging output to a log file. Once the linkworker terminates, it prints "SUCCESS" or "FAILURE" in the flag file.

The mommy routine starts all linkworkers in the background and then polls the servers to check on the flag file status. We poll instead of keeping our session open in case a transient network outage drops our network connection. As each server reports in, the mommy updates the job status and merges the remote log into the primary log.

The linkworker reports to Deployme which processes were actually shut down. After the link, Deployme only restarts those processes that were previously running.

Error Handling

A standard in Deployme is that user error should not result in a failed job. Instead, Deployme attempts to catch all user errors in the job creation step. This means that most of our users do not need to understand how Deployme works. These users just look for green boxes on the status page.

Errors are handled using Perl's structured exception handling. This system catches all expected and unexpected errors and gracefully prints an error message to the web page or log file before aborting.

All job execution actions are logged in text files that are stored for later review. The history table in the database tracks all jobs and stores the exception message if a job fails. Email is used for immediate notification of success or failure.

Automatic Rollback On Failure

Originally, part of our error handling strategy was to make individual tasks transactional. If we detected any problem with a task, we would set the state of the machine back to what it was before the task started. We found that automatic rollback on failure worked better in theory than in practice.

This policy only applied to platform and tool LINK tasks. Content LINK tasks rarely fail, and do not require any servers to be restarted. Failed PULL and BUILD tasks usually result from bad tags, and do not affect remote servers in any case. Failed PUSH tasks also do not affect remote server state because the PUSH logic copies the package to a temporary directory and renames it to the actual package name once the copy is complete. This prevents half-completed packages from appearing on remote hosts.

Initially, automatic LINK rollback seemed easy. The task is already separated into link, unlink, start,

and stop commands. Also, the parallel_mommy routine (discussed above) knows what tasks were completed before the task failed. The obvious rollback is to undo whatever was done and call it quits. If the rollback fails, exit immediately. If we are executing tasks in parallel, wait until they all report in and then initiate rollback for each one that failed, serially.

The reasons for automatic rollback on failure were straightforward. We did not want to suffer downtime if a job failed, so we needed the machine restored to a functional state. Also, we felt that LINK rollback was necessary to prevent database inconsistency. We were afraid that individual failures over a period of time could make the database progressively more unreliable.

We implemented the automatic rollback system, and it turned out to be such an operational failure that we tore the code out after several weeks.

The problem was that all of the reasons cited above were wrong.

Automatic rollback on failure became a major inconvenience for the network operations team. Instead of a job failing immediately so they could go in and fix it, they would be forced to wait while processes were restarted. The serial nature of the rollback made this maddening when multiple servers suffered similar failures.

Besides the delay, it turned out that operations did not want the machine restored to its original state. Due to our network architecture, it is not a problem to have a single machine out of service. After a successful rollback, the first thing that operations did was shut down the server that rollback had restarted.

Automatic rollback did not make the database more consistent. For many failed tasks, the database was inconsistent before the job began. Rolling back just went from one inconsistent state to the original inconsistent state.

We learned from this experiment. In our case, a fail-fast system results in shorter outages than a system that tries to fix problems on its own.

The Minimal Downtime Paradox

We found that some of our goals were not exactly what our Operations team wanted. All of the link scripts meet our "Seamless Activation" goal to get the servers back up and running as soon as possible. However, in regular maintenance scenarios that is not the usual Operations procedure. They often will take servers down for an extended time to perform other maintenance along with a software push. Also, some did not agree with the policy of "that which is on disk must be running."

As with rollback, we saw the paradox that restarting server processes automatically can result in longer outages, since operations will just have to stop and start the server anyway for their own maintenance.

In response, we added a feature to prevent server restarts from occurring during a LINK job.

War Stories

Deployme does not have any UI for performing reporting, but its database is enormous. The tables contain sufficient information to reconstruct every software upgrade Tellme has ever made since Deployme was introduced.

In several cases, we have used the database to ferret out information such as "When did this package reach the production servers?" and "Which modules were upgraded since yesterday?"

Deployme makes it so easy and fast to move to a previous package that our Operations team is rarely alerted when a bad package is released. Instead, the content producer quickly reverts to a good package and keeps the outage brief.

In many ways, Deployme has reduced the number of war stories we have to tell. The only major content-related outages since Deployme was instituted resulted from individuals going around the Deployme process and editing files directly on production servers. (Old habits are hard to break.) There really is no reason to do such a thing in the Deployme world.

Future Work

Despite Deployme's merits, the current implementation is creaking loudly. The code base, approximately 10,000 lines of code, has become difficult for us to maintain.

The maintenance problem centers around the conditionals sprinkled throughout the code to check for a particular set of module types. For example, the "execute link" subroutine performs a large conditional to figure out which Perl module to call. Another example is that the web UI is littered with module type checks to determine what options to offer when creating a tag.

Our solution for this is a concept called "services" which was introduced in [Finkel1997], but our usage is slightly different. Instead of a module type, each module will be associated with a list of target services where that module can be deployed. Servers will provide services instead of having a single server type. The services table will contain a reference to the Perl object that will provide the service-specific code, isolated from the Deployme core. Additionally, we will add a START and STOP task to the list of deployment tasks so service restarts are distinct from the LINK task.

One application of services is to allow Deployme to push to virtual web servers. Deployme only allows web content to go to a single location on a remote server. With services, we can push to the "developer" service on a web box separately from the "corpweb" service.

Our preliminary services plug-in implementation works very well. Because of this, we are moving all of the Deployme logic into objects descended from an abstract base class. For instance, CVS access will be abstracted under an SCS, or source control system, class. In this particular case, there has been some discussion about moving to Perforce [Perforce2000] from CVS, and we want to be able to accommodate such a switch with little pain.

All job execution steps and web UI are also being moved into subclasses. Once this is complete, the Deployme logic will be completely separated from the Tellme site logic.

Another problem with Deployme is that it has no concept of a physical location shared between two machines. We investigated changing the database schema to know the difference between physical and logical locations, but the result was extremely convoluted and hard to use. Instead, we plan to provide a mapping table that designates a single server as the master for a shared mount point. At job creation, we will map any requests for deployment to a specific server/service pair to the master server if specified in the mapping. Since the job creation logic removes duplicate steps, we can be certain that we will deploy to a shared location only once.

We plan to change the underlying database from MySQL to PostgreSQL [Lockhart2000]. PostgreSQL supports true referential integrity, which is necessary for the more complex Deployme2 schema. Also, PostgreSQL has transaction support, which makes error handling easier.

Security is another area where much work remains. We want to build a granular system that limits what a user can do. Deciding how fine to slice access control is a tough question. To start, we will probably create "roles" which have different capabilities.

We need a way to quickly copy a package to multiple servers on a subnet. We have some prototype multicast file transfer programs written, but we have not yet decided how to incorporate them into Deployme. In addition to transferring bits via multicast, we could also blast commands to many hosts at once.

Further enhancements include integrating Deployme into the system we use to initialize brand new machines. Also, we would like to add a "last-known-good" feature so that it is easy for operators to know what package is safe to roll back to.

Status

As of this writing, we are working on Deployme 2. We intend to integrate many of the conceptual changes cited in Future Work. Most of the technology upgrades, such as multicast, are not a priority right now. This is because we are more interested in adding

the ability to add technology than the technology itself.

Conclusions

We originally wanted to find an off-the-shelf tool for accomplishing the goals stated above. However, none of the tools we reviewed provided an end-to-end solution for package lifecycle management.

Deployme has greatly exceeded our expectations. While it began as a simple content manager, it has expanded to become the upgrade center for our entire server environment. It currently tracks a few hundred servers, and we believe it can track thousands as we improve the underlying technology.

The best thing Deployme has done for Tellme is something we never considered when writing up the goals. It has reduced our stress level.

Stress is not something normally considered when writing tools. Usually we focus on scalability, speed, ease of use, and other immediately evident goals. When all of these immediate goals are met and the tool is firmly inserted into your process, that is when you see the secondary benefits.

As Deployme began to encompass more teams within the company, our collective confidence grew and usage skyrocketed. Each day, Deployme processes 40 to 60 jobs encompassing hundreds of individual tasks.

Our development schedules are not held hostage to a release team. Instead, Deployme has engendered a release democracy, where even the newest employee is empowered to take over a module and start sending new packages through QA and up to production.

The best accomplishment of all is our most frequently asked question, "That's all I need to do?"

Author Information

Kyle Oppenheim is a software engineer at Tellme Networks. He graduated from Carnegie Mellon University with B.S. and M.S. degrees in Electrical and Computer Engineering. He is currently playing the role of Release Engineer, but is intently trying to automate his job away. Reach him electronically at kyleo@tellme.com.

Patrick McCormick is a software engineer at Tellme Networks. He graduated from the Massachusetts Institute of Technology with B.S. and M.Eng. degrees in Electrical Engineering and Computer Science. His current professional interests include voice recognition, computer telephony, and software deployment. His email address is patrick@tellme.com.

References

- [Cederqvist1993] P. Cederqvist, et al., "Version Management with CVS," http://www.loria.fr/~molli/cvs/doc/cvs_toc.html, 1993.
- [Dagenais1993] M. Dagenais, S. Boucher, R. Gerin-Lajoie, P. Laplante, P. Mailhot, "LUDE: A Distributed Software Library," *Proceedings of the Seventh Large Installation Systems Administrators Conference*, Monterey, CA, November 1-5, 1993, pp. 25-32.
- [Finke1997] Finke, Jon, "Automation of Site Configuration Management," *Proceedings of the Eleventh Large Installation Systems Administrators Conference*, San Diego, October 26-31, 1997, pp. 155-168.
- [Furlani1996] J. Furlani, P. Osel, "Abstract Yourself with Modules," *Proceedings of the Tenth Large Installation Systems Administrator's Conference*, Chicago, September 29-October 4, 1996.
- [Glickstein1996] B. Glickstein, "Managing the Installation of Software Packages," <http://www.gnu.org/software/stow/manual.html>, 1996.
- [Howard1988] Howard, John H., "An Overview of the Andrew File System," *Proceedings of the USENIX Winter Technical Conference*, Dallas, pp. 23-26, February 1988.
- [Lockard1998] J. Lockard, J. Larke, "Synctree for Single Point Installation, Upgrades, and OS Patches," *Proceedings of the Twelfth Large Installation Systems Administrator's Conference*, Boston, December 6-11, pp. 261-270, 1998.
- [Lockhart2000] Lockhart, Thomas, ed., "PostgreSQL User's Guide," <http://www.postgresql.org/docs/user/index.html>, 2000.
- [Manheimer1990] K. Manheimer, B. Warsaw, S. Clark, W. Rowe, "The Depot: A Framework for Sharing Installation Across Organizational and UNIX Platform Boundaries," *Proceedings of the Fourth Large Installation Systems Administrator's Conference*, Colorado Springs, CO, October 18-19, pp. 37-46, 1990.
- [Moore1999] Moore, Michael, "Publishing to the Web in a Distributed Environment," http://www.microsoft.com/backstage/bkst_column_10.htm, July 1999.
- [MySQL2000] MySQL AB, "MySQL Documentation," <http://www.mysql.com/documentation/index.html>, 2000.
- [Perforce2000] Perforce Software, "Perforce Documentation," <http://www.perforce.com/perforce/technical.html>, 2000.
- [Pierce1996] C. Pierce, "The Igor System Administration Tool," *Proceedings of the Tenth Large Installation Systems Administrator's Conference*, Chicago, pp. 9-18. September 29-October 4, 1996, pp. 9-18.
- [Sandberg1985] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, B. Lyon, "Design and Implementation of the Sun Network Filesystem," *USENIX Conference Proceedings*, USENIX Association, Berkeley, CA, pp. 119-30, Summer 1985.

- [Tridgell2000] A. Tridgell, "Efficient Algorithms for Sorting and Synchronization," http://linuxcare.com.au/tridge/phd_thesis.pdf, 2000.
- [Wall2000] L. Wall, "Programming Perl," 3rd ed, O'Reilly & Associates, Sebastopol, CA, 2000.
- [Ylonen1996] T. Ylonen, "SSH – Secure Login Connections over the Internet," *Proceedings of the Sixth USENIX UNIX Security Symposium*, San Jose, CA, pp. 214, 37-42, July 22-25, 1996.

Automating Request-based Software Distribution

Chris Hemmerich – Indiana University

ABSTRACT

Request-based distribution of software applications over a network is a difficult problem that many organizations face. While several programs address this issue, many lack features required for more sophisticated exports, and more complex solutions usually have a very limited scope. Managing these exports by hand is usually a time consuming and error-prone task. We were in such a situation when we developed the Automated Netdist program a year ago.

Automated Netdist provides an automated mechanism for system administrators to request and receive software exports with an immediate turnaround. The system provides a simple user interface, secure authentication, and both user and machine based authentication. Each of these is configurable on a package-by-package basis for flexibility.

Netdist is a modular service. The user interface, authentication and authorization are independent of the export protocol. We are currently distributing via NFS, but adding an additional protocol is as simple as writing a script to perform the export and plugging it into Netdist.

Introduction

At Indiana University, we have a large, extended, and heterogeneous Unix workstation presence administered by many different departments and organizations. The Unix Workstation Support Group (UWSG) is charged with supporting and advising the administrators of these machines. This includes offering a variety of systems administration classes, negotiating and maintaining site licenses with various vendors, distributing a large variety of unix software, and maintaining a Unix Users Group. We also provide traditional phone, e-mail, face-to-face, and extended contact support services.

At the time Netdist was developed the UWSG consisted of six members, five full-time and one part-time, distributed across two campuses. This is a lot of ground for such a small group to cover, and we are always looking for ways to increase the efficiency of our services without sacrificing quality or our customers' satisfaction. In early 1999, I was given the task of re-working one of our least efficient and least reliable services, the request-triggered distribution of software via NFS.

Software Distribution

The UWSG distributes a large and varied collection of Unix software, to a widely varied audience. Software export laws, license agreements, pre-existing vendor-supplied distribution systems, varying security requirements, and machines with slow or non-existent network connection all work together to fragment our software distribution into the following components. These components are the environment out of which Netdist grew, and their attributes greatly influenced the development on Netdist.

- Anonymous FTP
- Secure HTTP Access

- Open NFS exports
- Request based NFS exports
- Media Checkout
- Vendor Specific Distribution Tools

Anonymous FTP

The majority of the software we distribute is via anonymous FTP. We maintain a large (150GB+) site at <ftp.uwsg.iu.edu>. This FTP site is our preferred method of distributing files. This archive is available to the world and contains mirrors of most of the large Linux distributions, a CPAN mirror, a mirror of the Linux kernel archives, various freeware programs and utilities, and system patches. The benefits of this system are many:

- It supports many concurrent users, and is limited only by our current hardware and the university's bandwidth.
- It is available to anyone in the world with FTP access.
- It is easy to use, and the ratio of support incidents to usage is orders of magnitude smaller than any other service we maintain.
- It is efficient to administer, requiring little maintenance.
- It is a very well known system, and administration can be shared or passed on easily.

Unfortunately, anonymous FTP is not a viable tool for all of our distribution needs. Following are several of its limitations that have forced us to distribute packages using alternate methods.

- Access control is very limited. We maintain a distinction between Indiana University (IU) clients and non-IU clients. Anything beyond this becomes much more difficult to administer as the target audience shrinks and the complexity of administration increases.

- Security is poor. FTP has a long history of security deficiencies, vulnerabilities, and exploits. We don't make any sensitive data available via FTP, allow authenticated logins, or allow write access.
- We provide operating system media for system installations, and most operating systems aren't able to be installed over an FTP connection. Several Linux distributions support this, but we must also provide Solaris, HP-UX, IRIX, and other Unix flavors that do not.
- FTP sites cannot be mounted as local file systems. We offer several large software packages, and users might not have the free space to download a large software package, uncompress it, and then install it.

Secure HTTP Access

For software that needs to be firmly restricted to IU affiliates we use an Apache web server running SSL and an in-house mod_perl module that allows us to securely authenticate users against the University wide kerberos database. This system is useful for distributing smaller packages and we use it to distribute export-restricted security software, as well as IU-specific license codes for several software packages that we have licensed.

This service addresses the security failings of anonymous FTP, but doesn't address the concerns of installing operating systems or dealing with large software packages. In addition, the server installation and configuration is much more complex than anonymous FTP, and the overhead on the server is greater.

Open NFS Exports

We maintain open NFS exports of several Linux distributions for network installs and updates. This is a convenient way for administrators to set up new machines over the campus network. As open NFS exports lack any real security, we do not export any other software this way. Indiana University does block incoming NFS requests from outside the school network, so the exposure is relatively small.

Request Based NFS Exports

At the time I began this project, we were exporting Star Office (before it was purchased by Sun) and Sun's Workshop Compiler Suite through NFS on a per-request basis. This allowed us to control who had access to the packages, while giving our customers the convenience of installing these packages as if they were mounted locally as a CD image. Our customers were happy with this service, but we were managing it by hand which was inefficient and error-prone. I was assigned the task of replacing this service, when we learned that several new software packages would be added to it.

Media Checkout

We maintain a large collection of OS and application CDs that are available for checkout by request.

This has been a very labor intensive service for us, as we must manage the physical storage of these CDs, make copies of popular requests, keep track of who has checked out what, and gently remind admins when they've kept a CD too long. A separate project to resolve the inefficiencies in this process is currently wrapping up, with wonderful results.

This process is secure, and convenient for administrators to install from. It also provides those with only a modem or no network connection a means of acquiring software. It is, however, horribly inefficient, generally requiring 15-30 minutes of work per request. We must respond to the request, obtain the physical media, schedule a pickup time, meet the customer, check their identification, register the checkout into our tracking system, and then collect the software.

Vendor Specific Distribution Tools

We use vendor-specific software distribution tools from both HP and SGI. Both of these tools use remote procedure calls for transferring software patches, updates, and new packages. These tools are convenient for administrators on campus to use, and required little administration. Both of these tools are proprietary software, and limited in the software they can serve and the clients that can access them.

We use the IRIX Network Distribution server to distribute OS and Software updates as well as new packages. In order for a machine to be able to access the server, it must be listed in the servers.rhost file. This file tends to get rather large and outdated as time progresses, and must be pruned periodically.

The HP Software Agent Suite allows us to make HP depot files available over the network to HP machines on campus. Depots are automatically available to the whole campus, but implementing more granular permissions is a non-trivial task. The system uses a graphical interface to make patching and install software painless.

Request Based NFS exports: The problem

Under the old system, an administrator that wanted StarOffice or Sun WorkShop would send an e-mail message to our group account containing the software requested and the target machine's hostname. Upon receiving the message, we would verify that all of the relevant information was included, and if necessary request any information that has been left out. Then we would edit the NFS exports file by hand to include the client machine and a comment listing when the export expired. We then re-initialized the exports and notified the user that the export would be valid for three days. Once the export had expired, we removed the client hostname manually and then re-initialized the exports to force the expiration.

While this system worked, we knew it should be better. The ease of installation from an NFS export and the wide install base of NFS clients were nice, but there were problems with the system:

- Our response time to customer requests was inconsistent. Requests were only processed during business hours, and if we were teaching a class or were otherwise busy, it could take an hour or longer to process a request.
- The process was error prone, especially while editing the NFS export file by hand. Some problems we experienced were mis-typing hostnames, corrupting the NFS export file, and forgetting to re-initialize the exports. These were all easily and quickly resolved, but increased response times and introduced short service outages.
- Handling requests interrupted other tasks. While handling a request usually only took five to ten minutes, the interruptions could be difficult to deal with. We place a high priority on response time, so even important tasks would be interrupted to handle a request.
- The system was not scalable. The time required to process exports and the chance of error both increased linearly with the number of requests. With anonymous FTP adding an extra 100 users, would not be noticeable, but the same increase in NFS requests would have required us to open another full time position.
- Removing exports was problematic. It was easy to forget about an export after three days and not remove it from the exports file. There were also occasions where a hostname was added without comments, which made it difficult to determine if an export should be removed.

Requirements in a Replacement

After cataloging the strengths and weaknesses of each of our distribution systems, we constructed a list of features required in a replacement:

- Automatically process user requests.
- Automatically clear expired requests.
- Only answer requests from IU machines
- Must not require any proprietary software on the client machine.
- Must be easy to use and to administer.
- Logging must be complete.
- Be able to authenticate the user, preferably through our existing Kerberos database
- Any existing solution must be available free of charge.

With such specific requirements, we had serious doubts about finding an existing software package that met, or even attempted to address, our needs. We had been prepared to develop the solution from the beginning, and already had several ideas about how to implement the service. Before proceeding with this implementation, we did perform a limited search for existing software that met our needs.

We searched several common search portals, such as dejanews (now deja), yahoo, and altavista. Plodding through hundreds of results from various

search terms provided very few software distribution systems, and all of these were designed to face problems different from ours.

Failing to find an existing solution to our problem, we began work on an in-house solution. We wanted to maintain the benefits of a file system based export, and since only a handful of machines on campus had AFS or DFS, we decided to stay with NFS as the means of exporting the software.

We decided to leverage our experience with the secure web server interface to our kerberos database to provide secure authentication, and a web based request form. This gave us the ease of use of NFS, the security of HTTPS through user authentication, and nearly ubiquitous installation of our clients, i.e a web browser and NFS client.

With export requests and authentication handled through HTTPS, and the software exports done via NFS, we needed something to tie the pieces together. We looked at several different models before narrowing our selection down to either a system of setuid root C programs, or a client/server interface, with the web server acting as the client to a custom designed server. We implemented rough versions of both of these, and did some performance and security testing. From the results we decided to go with the client/server model.

A Brief Version History

The initial version of Netdist was designed to alleviate the immediate problem with exporting StarOffice and Sun WorkShop. We were eager to release the system, and as such were unable to implement several of the more ambitious planned features. The initial release in June, 1999, included the following features:

- Automated user authentication.
- Automated export initialization and expiration.
- Ability to include or exclude sub-networks of machines by IP number on a per-export basis.
- Variable export durations for different packages.
- Core API for implementing common methods.
- 24/7 availability, with immediate response.
- A client/server model with PGP based encryption.

The initial release went well, with positive user feedback. As it became certain that we would be adding ApplixWare and Island Office to our available software, we returned to Netdist development. In November 1999, we released a much improved Netdist with ApplixWare and Island Office added to the list of available software. The improved features included:

- Expanded API, and more encapsulated system overall.
- Ability to limit exports to specific users.
- System for managing export documentation.
- Ability to control exports from multiple machines from a single web interface.

- Ability to accept plug-ins for exports of protocols other than NFS.
- More efficient coding, faster response time on requests.

Since this revision, Netdist has been a nearly ideal service to administer. It has required intervention less than a handful of times, and the logs show continued use.

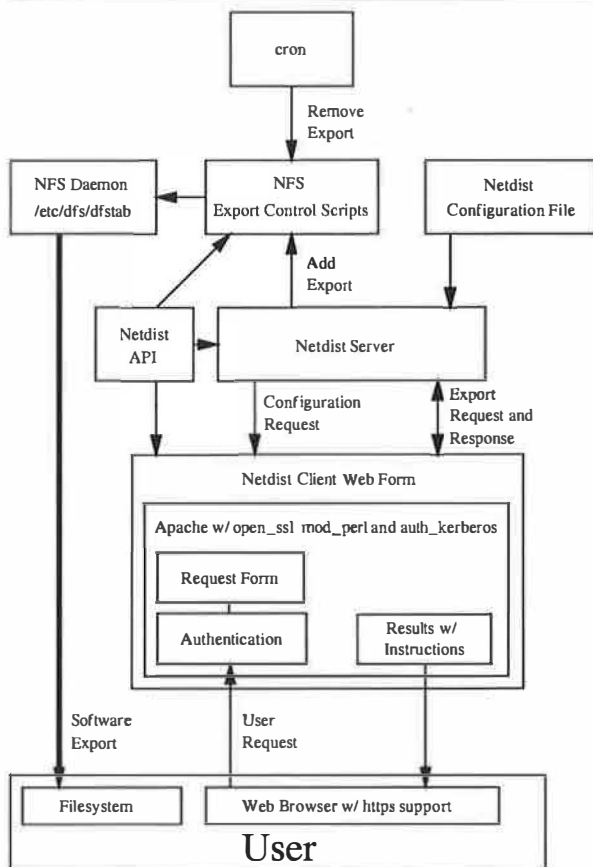


Figure 1: Information flow between Netdist components.

Architecture

The Netdist program is a system of four modular components that can each be upgraded and extended as needed with minimal effect on the other components. The components consist of an API, client, server (with configuration file), and export control scripts.

Briefly, the client is a web form that authenticates and authorizes each export request. If the request is validated, the client passes an encrypted request to the server. The server processes this request, and calls the appropriate export control script, depending on the export protocol of the software requested. The results of these exports are then returned to the client for presentation to the user. The API ties all of these sections together by providing common procedures and data structures. Upon startup, the server reads a

configuration file that defines the valid export types, export restrictions, and other settings described below, and shares this information with the client. See Figure 1 for an illustration of how these different components work together to manage software exports.

The API

The API is written in object-oriented Perl, and is available to the client, server, and access control scripts. The API is composed of six modules and contains any code that should be accessed from more than one component in order to increase efficiency and to ease communication between the different components.

The API provides procedures for accessing and manipulating the data structure created by reading in the configuration file. This allows each component of netdist to manipulate this data structure. The API also manages all logging in order to guarantee comprehensive and compatible log entries from events logged in different components. API functions are also available for creating unique temporary files and performing date comparisons.

The Client

The client, or user interface to Netdist is a collection of mod_perl scripts available on an Apache Server with SSL. We use a custom Apache module to authenticate users against our kerberos database, but other Apache authentication schemes could be used. Once the user is authenticated, they are presented with a list of the exports they are authorized to request. This list comes from the data structure which the server creates from the configuration file, and is later passed to the client upon request.

The user then enters the hostname he wishes to export the software to, and selects the desired software packages. The client performs taint checking on the data before processing it to look for illegal hostnames, and attempts to compromise the web server. The client then makes sure the hostname entered is an IU machine, performs a reverse DNS lookup to make sure the hostname and IP number match, and confirms that the selected exports are valid for that host.

If these conditions are met, the client opens a connection to the server and submits an encrypted request for the approved exports. The client waits for a success or failure response from the server and returns a message for any error that may have occurred. Otherwise, the client returns success, with links to instructions for accessing the exports, and an expiration date.

The Server

The server is a long-running Perl daemon that must run as a user privileged enough to manage the exports. In our case, NFS is used and the daemon runs as root. At startup, the daemon processes the configuration file, and then listens for connections on a UNIX or TCP port, depending on how it is configured. The server uses a PGP authentication scheme to ensure

that it only processes requests from the client. There are two valid requests a client can send to the server. This first is for a copy of the configuration data structure, and the second is to export a software package. Configuration requests are responded to, and export requests are mapped to the appropriate export control scripts, and results of the export are returned.

The Export Control Scripts

Each type of software export requires its own set of export control scripts. Each set consists of one script the server calls when requesting a new export, and a second script to clean up expired exports, which should be run by cron or another scheduling utility. In the case of NFS, the exports controlled by Netdist must be specially formatted to allow the control scripts to process them. Other exports can be listed normally after the final Netdist export. On a Solaris machine the formatted entry looks like this:

```
#@work50          begin Island exports
#host1.indiana.edu user1 2000/09/18
#host2.indiana.edu user1 2000/09/18
#host3.indiana.edu user2 2000/09/19
##work50          end Island exports
share -F nfs -o ro=host1.indiana.edu:
                 host2.indiana.edu:host3.indiana.edu:
                 filler.indiana.edu /is/netdist/ws50
```

A comment entry is added with each export containing the hostname, the username of the requester, and the date of the export. The comments not only allow the expiration scripts to determine when an export has expired, they also make the export file more informative to human readers.

Configuration

We designed the Netdist program to be highly and easily configurable. From working with many Unix programs, we felt that a simple yet powerful plaintext configuration file was the best way to achieve this. The format of the configuration file, and the API calls for processing it, were the first sections to be implemented.

We had an initial list of values to store in the configuration file, which grew as we developed the other sections. We attempted to extract every configurable value to this file, but there are a few values that we missed, and will extract in the next version of Netdist. The Netdist configuration file defines the available exports and their properties expressed as key-value pairs. The valid keys are shown in the subsections below.

export

This value is a short string that uniquely identifies the export within the system. It must be defined for each export, and is only displayed in the logs. Examples: (star51, isl60)

text

This value is a text string that identifies the export to a Netdist user. This should contain the full

title of the software package, the version number, and the platforms on which the package is valid. This parameter must be defined for each export. Example: ('StarOffice 5.1 (Solaris|Linux)')

type

This value defines the type of export the package is. Currently the only valid choice is nfs, but if other export protocols are added, this key will be used to determine which scripts to call to perform the export.

host

The host value sets the host on which the export is located. The host must have a configured and running Netdist server. With this value a single web interface can manage exports from multiple servers.

port

This value defines the port on which the appropriate Netdist server will be listening. This allows Netdist to be easily introduced into environments without worrying about any given port being free.

duration

The duration value defines the number of days an export is valid before being expired.

manual

This value defines the name of the file to link to when providing instructions for acquiring and installing an export once it is approved. Instructions should be in HTML format.

users

The users value defines the list of users allowed to access this export. A value of "*" will allow all users to export the software. Alternatively a list of usernames can be provided. Only users able to export the software will see it as an option.

allow

This value defines IP ranges to which the software can be exported. Each entry is expressed as a subnet mask, and multiple entries are allowed for each export.

deny

This value defines an IP range on which the export is not permitted. This can be used to remove part of a previously allowed network, including hosts and subnets. The syntax is the same as the allow parameter, and again multiple entries are allowed.

Configuration: An Example

The first export defined in the configuration file is a default. Any keys that are unassigned in subsequent export definitions will use the default value. Below is an abridged configuration file, and an explanation of the entries.

The First line declares the beginning of the default entry. The next few entries declare that the default protocol will be NFS, and indicate the the host and port on which to look for the server on. Next the

file specifies that by default all users will be able to access an export, and then gives the valid IP ranges for exporting software.

```

Default
type=nfs
host=netdist.domain.edu
port=888
users=*
allow=111.111.
allow=111.112.
export=soft1
text='Software Application \
      1.1 (Solaris, Linux)'
manual=softappl.html
allow=111.113
allow=111.111
deny=111.111.12
export=soft2
text='Software Application 2.1 (IRIX)'
manual=softapp2.html
deny=111.112.1

```

The export key defines the beginning of a new export. The value given becomes an internal identifier for the export and must be unique. The text and manual definitions function as described above. The two allow statements are combined, and they then replace (not expand) the default allow value for this export. The deny is then masked over the new allow value.

The second export defined is similar to the first, but different in a few key ways. First, the export value is a different string. This is essential as each export string must be unique within a netdist system. Also, since no allow key is defined, the default allow values are used and the deny value given is masked over them.

Below is a partial dump of the data structure created when the configuration file is read in. This structure is then passed on to the web client so that it can only lists the exports a user has access to and to make security checks against the requests before passing them on to the server. This dump was created with the Data::Dumper Perl module.

```

$VAR1 = bless( {
  'star52' => {
    'deny' => [],
    'users' => '*',
    'type' => 'nfs',
    'port' => 2110,
    'text' => 'StarOffice 5.2'.
              (Solaris|Linux)',
    'doc' => 'star52.html',
    'duration' => 3,
    'host' => 'server.indiana.edu',
    'allow' => [
      '^111.12.',
      '^111.14.',
      '^111.15.'
    ]
  },
  'default' => {

```

```

    'deny' => [],
    'users' => '*',
    'type' => 'nfs',
    'port' => 2110,
    'text' => 'default',
    'duration' => 3,
    'host' => 'server.indiana.edu',
    'allow' => [
      '^111.12.',
      '^111.14.',
      '^111.15.'
    ]
  },
  'work50' => {
    'deny' => [],
    'users' => '*',
    'type' => 'nfs',
    'port' => 2110,
    'text' => 'Sun WorkShop 5.0 (Solaris)',
    'doc' => 'workshop50.html',
    'duration' => 7,
    'host' => 'server.indiana.edu',
    'allow' => [
      '^111.12.'
    ]
  }
}, 'Configuration' );

```

Security

Security was a priority throughout the development of Netdist. We designed the architecture around several basic security tenets.

- Users need to be authenticated before accessing the service.
- Need to securely process NFS requests (which require root access), through our web server which runs under an unprivileged www account.
- Logging must be thorough and structured to make potential security problems highly visible.

Authentication

We have experience using apache with open-ssl along with with an internal mod_perl module to authenticate web connections against IU's kerberos database. These components had worked well for us in the past, allowing us to authenticate any IU user, and ensure that their username and password were safely transmitted to the server. Once authentication has been performed, Netdist works with the user through an abstract interface. Other apache authentication tools can be used, and the resulting data can be structured to work with the abstract database.

Request Processing

We required a method of responding to requests that was secure against local users as well as network based attacks. We investigated several options, including setuid programs, before deciding to use a client/server interface. They allowed us to securely

make the transition from web server to root permissions, and seemed more expandable than other solutions.

After the web form processes the request, it makes a socket connection to a server running as root, and transmits the username, machine name, and export request. The server then processes and logs the request and sends the results back to the web form for display to the user. To ensure a robust service, Netdist supports both TCP and UNIX sockets for its client/server communications.

We must ensure that the server only replies to requests from authorized clients, that it encrypts any data sent over the network, and that the server handles malformed connections properly. Using UNIX sockets for communication simplifies this somewhat, as you only worry about attacks from the local machine, and can use file permissions to limit access to the socket. For this reason, UNIX sockets are preferred for Netdist, unless you need the ability to control exports on a foreign machine, in which case TCP sockets are required.

We use PGP to secure communication over the sockets. We generate a public/private key pair, assign the private key to the root account, and place the public key in the web server account's (www) keyring. The public/private nomenclature of PGP encryption isn't truly appropriate, as the public key is also private, and known only to the www account. No other copies of these keys exist, and both files can only be read by the owner. Neither of these accounts uses PGP for any other purposes, and the key pair is not used in any other way.

In effect, this configuration allows us to encrypt and sign the message with a single key pair. Since the www client is the only entity with access to the public key, any message encrypted with it has been signed by the client. Initially, we used a second public/private key pair to sign the encrypted message in the traditional PGP form. However, we eventually realized that the second key pair didn't increase security. Our www client's private key was no more secure than the server's public key, and only served to complicate the communication protocol.

This encryption/signing model relies on keeping the two keys a secret. This is a significant concern, as one of the keys is owned by the www account. Web server accounts are common targets of crackers as they often provide easy access to a machine. Not only must we worry about an attacker exploiting an insecure cgi script or server setting, it is also trivial for any user able to host cgi scripts on the web server to write a script that will print the keyfile. To reduce this risk, only members of the UWSG have accounts on the machine that hosts netdist, and only the www and root account can host web pages on the secure web server. Very few web scripts are run on this secure server, and all are closely examined, and perform strict taint checking on any entered data.

Each message sent by the client is encrypted with the public key. The server forks off a child process for each incoming message. It then goes through a series of steps to confirm the validity of the message. If any of these steps fail, the message and test failed are logged and the child process drops the connection and terminates before proceeding to the next step. These steps are completed in the following order:

1. The server confirms that the message is PGP encoded, and does not contain any additional text or data.
2. The server decodes the PGP message to ensure it was sent from the client.
3. The server performs taint checking on the command to ensure that it is a valid netdist command with no special characters, shell escapes, etc.

Once these tests are passed, the data is passed as a parameter to the appropriate export control script, and logged. At no point is code received by the server executed by perl or the shell. These precautions are in place to limit the impact of any successful attacks against the encryption/signing scheme. First and foremost, arbitrary programs can not be run as root, and the aggressive logging should allow us to identify any successful attacks and close the service until the problem is resolved.

Logging

Netdist maintains three different log files for readability. The first file logs any errors that occur during the export request process. This is the log where attacks on the system will be documented, as well as any problems that might be affecting the system. As such, it is important to closely monitor this log.

The second log file logs the time, username, supplied hostname, and requested exports of every request. This log gives a comprehensive listing of the requested exports, and can be used in conjunction with the error log to look for suspicious access patterns.

The final log records successful export requests, including the username, hostname, and timestamp of the request. This file can be used to generate many different statistics on usage. It can be coupled with machine and user databases, to answer questions such as "How many requests were made by graduate students last semester?", or "How many requests for Star Office have been made from the Chemistry Department?"

Distributed Netdist

The Netdist program supports a distributed model, where a single web client can control software exports from multiple machines via multiple servers. This is useful for controlling multiple exports that can't be hosted on the same machine. This model requires a single master machine which contains the netdist configuration file, an instance of the Netdist

server and the Netdist API. Additional machines that will export software are slaves and must contain an instance of the Netdist server, the Netdist API, and the export control scripts for each type of export it offers. The web client can be installed on any machine, and acquires its configuration from the master server. See Figure 2 for an example of how multiple machines can work together in a distributed Netdist environment.

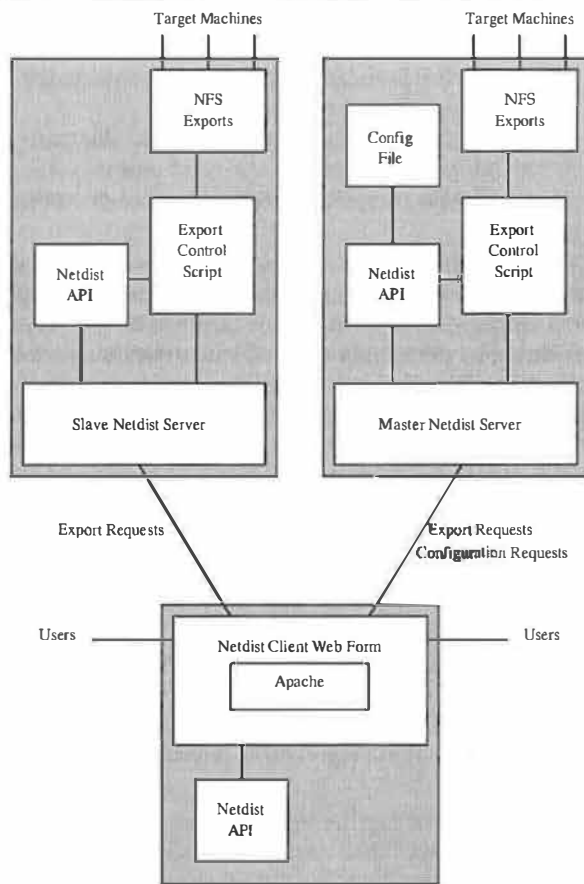


Figure 2: A sample Distributed Netdist implementation.

Portability

Currently, Netdist could be ported to other sites and platforms with a modest amount of work. It requires a host with Perl 5 and some modules from CPAN, PGP, cron (or some other scheduling routine), and an instance of Apache with at least `mod_perl` and preferably a module for secure transactions. The NFS export control scripts have been written for Solaris, but could be modified to work with the syntax of other Unix flavors.

The Perl modules and several scripts do have host or port specific information coded into the script, in a few places. Each of these instances is documented, and easy to update. In the next version of Netdist these values will be extracted to a configuration script to increase portability.

The Future of Netdist

The development of Netdist has slowed, as I no longer work for the UWSG, and have less time to work on it. However, the following items have been completed/are being worked on:

- Scripts to allow Netdist to control https access via `.htaccess` files
- Scripts to allow Netdist to control DFS and AFS exports via group membership
- Migrating from PGP to SSL for encrypting communication between the client and server. This is dependent on SSL support for perl through `Net::SSLeay` becoming more robust, or me re-writing the client/server in C.
- Modifying the user interface to support many more available packages via a tiered menu system.

The flexibility of Netdist is important to us, as preferred protocols and programs can change quickly. Netdist allows us to maintain a single, familiar point of contact for customers, despite changes in the way data is moved. For example there has been a push at IU for DFS and AFS in the past few years, and Netdist is ready to work with controlling access to software available on these systems.

Availability

Netdist is still pre-alpha in that we haven't done much work to ease installation, and we would like to incorporate some of the features mentioned above to increase its usefulness. As such, Netdist is not yet widely available. If you are interested in Netdist, please contact the author for the software location, and help with setting it up.

Acknowledgments

I would like to thank the people who worked with me in the UWSG at the time I was working on the Netdist project for their support, suggestions, and help with testing the system. In particular, Dick Repasky initially encouraged me to submit this work to LISA, and worked with me extensively to revise the first draft of this paper.

I would also like to thank the current members of the Data Storage Services Group for supporting my work on this paper, even when it interfered with my current job responsibilities.

Finally, I would like to thank those that helped me review and revise this paper. Their patience and willingness to repeatedly suffer my writing was instrumental in finalizing this paper, and is greatly appreciated.

Author Information

Chris Hemmerich <chemmeri@indiana.edu> works as a System Administrator at Indiana University (IU). He first became affiliated with the University as a student in 1993. Since then he has received a BS in

Biology, and is working on a Masters in CS. In 1995 he began working for IU as a lab consultant with the University Information Technology Services (UITS) division. From there he worked through various positions, including UITS Education Program Instructor/Developer, UITS Knowledge Base Programmer/Editor, and Unix Workstation Support Group Unix Systems Specialist, where he served as the primary HP-UX support contact for IU, and developed the Automated Netdist system. He currently holds a position with the Distributed Storage Services Group, where he administers IU's DFS and AFS infrastructure, while assisting with the administration of IU's HPSS installation. In addition to the e-mail address above, Chris can be reached by U.S mail at Indiana University; 2711 East 10th Street; Bloomington,IN 47408

References

- [1] *CPAN documentation for Perl modules*, <http://www.cpan.org>.
- [2] Christiansen & Torkington, *Perl Cookbook*, O'Reilly and Associates.
- [3] Stein & MacEachern, *Writing Apache Modules with perl and C*, O'Reilly and Associates.
- [4] P. Zimmerman, *The Official PGP User's Guide*, MIT Press.
- [5] Perl man pages, especially perlipc.
- [6] The Apache Website, <http://www.apache.org/>.
- [7] B. Wong, *Configuration and Capacity Planning for Solaris Servers*, Prentice Hall.
- [8] The Open SSL Project, <http://www.openssl.org/>.
- [9] H. Stern, *Managing NFS and NIS*, O'Reilly and Associates.
- [10] *Distributed File Service Administration Guide and Reference*, IBM.
- [11] The WU-FTPD Website, <http://www.wuftpd.org/>.

Use of Cfengine for Automated, Multi-Platform Software and Patch Distribution

David Ressman & John Valdés – University of Chicago

ABSTRACT

Good UNIX system administration practice includes among its many tasks the proper configuration of system files, installation and maintenance of third party software, and maintenance of system security, including regular updates of operating system (OS) patches. For a small number of systems running only one or two OSes, keeping up with these tasks isn't too difficult. However, as the number of systems and OSes increase (and the number of staff remains constant), these chores can quickly become overwhelming.

This paper describes our planning, development, and deployment of a system that provides automated software distribution, patch installation, and OS configuration through the integration of GNU cfengine [Bur95], MySQL [MySQL00], and a few custom written Perl scripts. It is meant to be less of a tool description and more of a discussion about the various aspects of designing a multi-platform software and patch distribution system, and the benefits of integrating those systems into a configuration management system such as cfengine. Designing and developing our system has been a time-consuming endeavor, but it has proven to be well worth the effort.

Background

Our network was out of control. In our department, we have over 100 UNIX systems running more than half a dozen UNIX based OSes (more than a dozen when counting different OS versions). The vast majority of the systems share a similar role, but they are all configured slightly differently to suit their particular users' needs. With only two systems administrators, it was extremely difficult to handle the day to day maintenance for each of these systems. We fell so far behind that the majority of our days were spent merely fighting fires. Because of this, we had very little time to respond to requests for software updates or new software installations and even less time to make sure that all of our systems were running at the current OS patch level. This left us with two large problems:

- Most of our systems had very old copies of software. Since there was so little time to spend upgrading software, we would only upgrade or put new software on a machine when a user would specifically ask for it. It's not hard to imagine what this led to; we ended up with a number of different versions of the same software all installed slightly differently across our systems. This made upgrading software a much more difficult task than it had to be, so unless there was a specific need for it, software wouldn't get upgraded at all.
- Most of our systems were unpatched against known security problems. The chore of manually applying patches to over 100 systems one by one every time a new patch report comes out is enough to give most systems administrators nightmares. Because of the great effort involved, most of our computers only had

whatever patches were current at the time of OS install and whatever patches were applied to the systems after mass break-in attempts.

It was clear to us that we needed some way to manage the distribution and installation of software and patches for our systems. We felt that eliminating these two problems more than justified however much time would be spent developing a new management system.

Requirements for a New System

Over the period of several days, we brainstormed about what our ideal system should include and came up with the following rough list:

- Software distribution and management
- Patch distribution
- Centrally controlled configuration
- Ease of use
- Low cost
- Security
- Flexible host configuration
- Centrally controlled "pull" of software and patch distribution
- Portability
- Autonomous operation

Software Distribution and Management

Our system needed to be able to handle the distribution, installation, upgrade, and removal of most or all of the software not supplied by the OS vendor that was needed on our computers. Additionally, it would be useful if our system could track information about installed software packages, such as a list of all files included in a software package, the original source of the software, and how it was compiled.

Patch Distribution

Our system needed to be able to handle the distribution and installation of all vendor supplied OS patches. It should also be able to handle any post-installation steps required by the patch, such as restarting a patched daemon. For maximum security, it would need to make sure that all of our computers were as up to date with their specific OS's patch list as possible.

Centrally Controlled Configuration

As much of each machine's configuration as possible (e.g. automounter configuration, printer setup, firewall configuration, syslog configuration, etc.) should be initiated and tracked by a central system. Likewise, the list of all software and patches which should be installed on each machine should be managed by a central system. The exact configuration state of each machine in the system should be reproducible in the event of a hardware failure or any other event that would require a fresh OS install.

Ease of Use

Our system had to be easy to use once placed into production. A system that was complex and difficult to use wouldn't be much of an improvement over manually maintaining our systems and would likely go unused. The system should also be easy to maintain and require a minimal amount of work to keep operational. The less work it required, the more time we could devote to other projects. The more time we could devote to other projects, the more productive we could make our users.

Low Cost

Our system must be inexpensive. We have a minimal operating budget and can't afford multi-host licensing fees and yearly software maintenance costs. This unfortunately ruled out most commercial software options.

Security

Security was a primary motivation for our undertaking this project. Our new system must allow us to achieve a higher level of security across our network than we previously had. This necessitated that our system be able to maintain a current OS patch level on all of our users' computers. It should be able to quickly upgrade software across all systems whenever security bugs were found in third-party software. Any central servers and processes used by our system should also be as secure as possible.

Flexible Host Configuration

We can classify our machines as belonging to one or more specific groups (i.e., NFS servers, laboratory workstations, members of specific research groups, etc.). Depending on what group(s) a machine belongs to, our management system should be able to install specific software packages and make specific operating system configuration changes. For example, machines grouped as laboratory workstations may

need additional data analysis packages installed on them, while only machines belonging to a specific research group will need a print queue defined for a printer which belongs to that research group.

We also needed to be able to manage software and OS configuration on a host by host basis. Our system should also be able to customize a machine's software and OS configuration beyond the configuration it receives by virtue of its group classification.

Finally, it should also be possible to customize a machine's configuration files based on any particular third-party software packages or vendor supplied patches that are installed. For instance, for machines with the Apache web server installed, we need a way to manage the server daemon's configuration and log files.

Centrally Controlled "Pull" of Software and Patch Distribution

The software and patch distribution, while configured centrally, should work on a system where the patches and software are "pulled" from a central server, rather than "pushed" by the server. By having "smart clients" and a "dumb server," we hoped to minimize the amount of damage that could be done to our system by the temporary loss or compromise of any one host. In our ideal situation, any or all of the functions of the server could be moved to different computers with a minimal impact to the system.

Portability

Our system must be portable. Our needs require it to run across multiple OSes, including SunOS 4.x, SunOS 5.x, IRIX, AIX, Digital/Tru64 UNIX, OpenBSD and Linux (including multiple architectures, such as Intel and Alpha). Fortunately, we do not need to support any non-UNIX OSes at this time.

Autonomous Operation

Most importantly, our system needed to do all of the above with as little intervention from us as possible.

Home-grown or Public Domain Software?

At this point, the only thing we knew for sure about our new system was that it would have to have three major parts to it: software distribution, patch distribution, and configuration management. The question we next asked ourselves was how much existing software could we use, and how much would we have to write ourselves? Our problems certainly weren't unique, and we didn't want to reinvent the wheel. Likewise, we aren't software developers and didn't have the time to write a complete system from scratch, so we wanted to make use of as much pre-existing, maintained software as possible.

Software Distribution

We spent a few days looking around on the web for software distribution systems but were only able to find a handful of references. When we took a more

careful look at the few systems we were able to find, we noticed that most of them relied on package directory trees being pushed out from a server (rdist style), copied over from an NFS server, mounted from an NFS server with symlinks set up in the client's local directory structure, or some combination of the three (Xhier [Sel91], Depot [Col92], Depot-Lite [Rou94], GNU Stow [Gli96], opt_depot [Abb97], SEPP [Oet98]).

We had already ruled out a server pushed distribution system, so the only other choices for existing systems were distribution through AFS or NFS from a master server. AFS was not an option we could consider because it was not available for all of our target platforms nor was it freely available for most of our platforms. While NFS may have been an easy choice for the authors of these other distribution systems because of a pre-existing NFS architecture, it was not an easy choice for us. We had made very little use of NFS in our department for software sharing or distribution.

Historically, we had always avoided running software mounted from an NFS server in favor of each machine having local copies of all of its software. In doing so, we minimized the amount of damage the loss of any one computer or the loss of our network could do to any other computer. We're employed to keep our users as productive as possible by allowing them to think about astronomy and astrophysics and not about whether the NFS server that is holding the software they need to use is available. Inevitably, computers and routers will crash or have to be brought down for maintenance; we didn't want to use a distribution system that would cripple all of our users' computers through the unavailability of a single server. Having our software distributed from or mounted on an NFS server would increase our dependence to a specific server more than we felt comfortable with.

Additionally, given our requirement to support multiple operating systems and architectures, using an NFS-based distribution system would have required that we either maintain a separate NFS server for each OS/architecture combination or maintain a complicated, multiple-architecture directory structure on a single NFS server. Lastly, we have software which must be installed locally on each system, such as software containing kernel modules and security related software (e.g. tcp-wrappers, ssh, etc.); if we used an NFS-based system, we would still need something else to manage the locally installed software.

After ruling out server push and AFS/NFS, we were unable to find any free software distribution systems which we could put to use in our department. It was clear that our best option was to write our own software distribution system.

Software Packaging

We needed our system to handle the installation, removal, and upgrade of 100 or more different third-

party software packages. Since we had ruled out a network filesystem based distribution method, we immediately realized that we would have to use some sort of packaging system to get our software from our distribution server(s) to our client computers.

We came up with the following three options:

- Use the native package format for each OS.
- Create a home-built package format using shell scripts and tar.
- Use an existing, multi-platform package format.

Of these three, we immediately ruled out using each OS's native package format for software distribution. While it (arguably) might have allowed for the most trouble-free integration of the software distribution system with the client computers and their OSes, it also would have been the most work to set up and maintain. Since each OS has its own unique packaging format, every piece of software we wanted to distribute would have to be packaged up in as many as eight completely different ways. The last thing we needed was to make packaging more complicated than it had to be. Besides, we had had enough experience with some of the package formats to know that a few of them were complicated at best and downright obtuse at worst.

A home-built package format using tar and shell scripts would certainly be the easiest to put together, since every OS we were using had tar and a Bourne-compatible shell. Given that, no additional software would have to be added on the computer in order to install, upgrade, or delete packages. All we would need to do is:

- Write shell scripts to install, remove, and query our available packages,
- Compile each software package once for every OS for which we wished to have the package available,
- Tar up the software into our package format,
- Finally, put the tarball up on our distribution server(s) to be pulled in by our clients.

It did have one drawback; we'd still need to get the package from the server to the client. Since we'd ruled out NFS, the only reasonable ways we could think of to get the package to the client over the network were HTTP or FTP. That alone wouldn't be enough to stop us from using a home-grown package manager. We could install a program like GNU wget or NcFTPget when we loaded the shell scripts on the client. It would be bare-bones and wouldn't have a lot of fancy bells and whistles, but it wouldn't be a bad solution.

When looking around for a Few Good multi-platform package formats, we came across a couple lesser known ones that looked like they might be able to work, but the clear leader in that field is the Red Hat Package Manager (RPM) [RPM00]. Aside from already having been ported to every operating system

on which we would need to run it, RPM has a very active development group, a very broad user base, and has been thoroughly tested.

RPM also extends beyond just a packaging format into a development environment that can control the entire packaging process from the compilation of software from source code on up to the installation of the packages. It also has the added benefit (or hindrance, depending on how you look at it) of extensive package dependency awareness, and has a built in FTP client so that it can retrieve packages from a remote FTP server and install them in one step.

Building RPM packages would require more initial work in that we would have to write a package specification (spec) file for each package. The spec file contains all the information that is needed to compile, install, uninstall, and upgrade the package for every OS for which the package would be available. However, we didn't think this was necessarily a bad thing. It would enable us to document from where we got the package's source code (not always an easy thing to remember or find). It would also very clearly show us every step we would need to take in order to compile and install the software (again, not always an easy thing to remember). Additionally, once we had written the spec file, updating packages to newer versions of the software would usually be very easy. Often, all that's required to build a new RPM package when updated versions of software are released is a one-line change to the package's spec file followed by an "rpm -b -a package.spec." The RPM manual [Bai99] contains an excellent description of the entire process of building an RPM.

The only problem we saw with using RPM would be the initial installation of RPM itself. We decided that if we were going to use RPM, we would build an RPM package in each OS's native package format, install that package on each system, and let RPM install all the additional software packages.

While the tar and shell script option would certainly have sufficed, we felt that the added functionality of RPM (especially the FTP client) was worth the effort of learning how to write the RPM spec files and building RPM packages for each OS. The tar and shell script option also would have required that we reinvent much of the functionality already in RPM. Therefore, we felt our best option was to write our software distribution system around the Red Hat Package Manager with packages distributed from an FTP server.

Patch Distribution

However disappointing the lack of information on available software distribution systems was, the lack of information on patch distribution systems was doubly so. We were unable to find any information on multiplatform patch distribution and installation systems. In fact, the only useful information we were able to find about any kind of automatic patch download and installation software was a reference to a program

named PatchReport in an article in the October 1997 issue of ;login: [Sin97]. PatchReport was a Solaris-only Perl script that would compare the current patch state of the machine it was being run on against Sun's patch cross-reference file. PatchReport would download any patches that were missing and install them. Since a large fraction of our computers were running Solaris, this was a pretty good start, but since it was a Solaris-only utility, it was clearly not going to be our multi-platform patch distribution and installation utility.

We were unable to find any such utilities for IRIX, Red Hat Linux, AIX, or any other OSes we would have to support. If we wanted a true multi-platform patch distribution and installation system, it appeared we would have to write one ourselves.

Since patches are supplied by the OS vendor or development team, there was no way to use a common patch format. Our only choice would be to write a system that was intelligent enough to recognize what OS it was running on and download and apply the right patches. Since we were already going to use an FTP server to distribute our software, we decided to use an FTP server to distribute the OS patches as well.

Configuration Management

When we started the search for configuration management software, we immediately found hundreds of references to a program named cfengine. After a cursory glance at the online documentation we found on the cfengine web page, we decided that this software package was definitely worth a closer look. We downloaded the cfengine reference manual [Bur99] and went home for the weekend to read it. The more we read, the more we realized that cfengine was a perfect choice for our configuration management system.

Cfengine would allow us to control every aspect of a machine's configuration that we thought would be necessary and then some! A complete description of cfengine is beyond of the scope of this paper, but briefly, cfengine will let you manage:

- Copying of files, both locally and remotely
- Editing of files
- Creation, removal, and maintenance of symbolic links
- Filesystem access control lists (ACL)
- File and directory permissions and deletions
- Filesystem tidying
- External command execution
- System and user processes

among many other things.

All of cfengine's actions can be conditionally applied based on whether or not certain "classes" (cfengine's name for "groups") are defined. Cfengine provides predefined classes based on the OS a given system is running, the hostname of the system, the day of the week, etc., and also allows you to define your own classes. Cfengine can also define classes

dynamically at runtime, so that classes can be defined, for example, only if a certain action was carried out. Given cfengine's actions and ability to group actions based on class, it was clear that we could use cfengine to manage system configuration which applies globally to all of our computers, to any individual computer, or to any predefined or dynamically defined group of computers, as we required.

As an added benefit, cfengine has a framework for allowing users to write their own modules in any programming or scripting language and merge them into a stock cfengine run. Through this, you can add nearly any functionality to cfengine without having to actually modify cfengine itself. Modules can use and define classes as well, giving them the same control and flexibility as cfengine's builtin actions. Because of this capability, we decided that we would implement our software and patch distribution system as cfengine modules so that we could use cfengine's class mechanism to manage configuration based on installed software and patches.

Summing It Up

We now knew that our solution would include a home-written software distribution system built upon RPM software packages, a home-written patch distribution system, and cfengine for configuration management.

Drawing Up a Rough Design

Having decided on the features we would want and the software we would use and create, it was time to figure out what we would need to write into our software so that it would meet our requirements.

Class Definitions

As we've mentioned, all of our computers can be classified as being part of one or more larger groups that share a common functionality. There are personal workstations, laboratory workstations, data servers, computational servers, etc. The systems can be further divided into smaller groups; there are groups of machines that are used for a specific research project, share a common user base, need the same software, etc. Depending on which group a machine belongs to, it will have different software installed and its configuration files will be different.

To handle this varied configuration, we would simply make use of cfengine's class mechanism. For this, we needed to define classes which reflected the grouping of our systems, and then create cfengine input files which would associate our systems with our classes and run whatever actions are necessary in order to configure the systems as needed.

Central Configuration and Control

We needed our system to centrally manage all system configuration. Having all configuration information resident on a central server makes it easy to change configurations for multiple systems with a

single edit, minimizing mistakes and eliminating inconsistencies between systems. Cfengine naturally lends itself to central configuration. In the most common method of cfengine use, master copies of cfengine's input files (the files that tell cfengine what commands to perform on the computer) reside on the cfengine server where they are pulled in by the client systems through cfengine's internal network file copy protocol.

We also needed our system to centrally manage all of the available software and patch information for all of the OSes we use with easy expandability for any future OSes we might need to support. Every single system needs to be able to talk to the central system and be told exactly what software packages should be installed and what patches should be applied.

We came up with two different possible ways of distributing this information:

- Have each client pull in a group of text files that describe the available software, each machine's software configuration, and each OS's patch configuration.
- Have each client query a relational database to obtain its software profile and patch information.

A text file option would be the easiest to set up. However, when we did the math to figure out how much data would be passed around in those files, it became clear to us that keeping track of over 100 software programs on over 100 machines would require more than 10,000 different entries in our text file – for the software subscription information alone.

Having two people maintain this text file was a recipe for disaster. No matter how careful we were, we were bound to make syntax mistakes and typographical errors. The more entries we had in this file, the harder it would be to track down any mistakes. While we could probably keep it under control with some difficulty, it was very clear that this option would not scale as the number of hosts and software packages increased. Maintaining a separate file for each computer with its own software information would have been just as bad.

Having this information stored in a database was clearly the better choice. With properly written front-end scripts to maintain the database information, it would be very easy to keep the database relatively error-free, and make the inevitable errors very easy to find and correct. In addition, most databases are designed with scalability in mind and could hold far more detailed information for far more hosts than we would ever need to support.

Hence, we decided to store all the software and patch information in a central database. Of the free database packages available, we chose MySQL because we had had some familiarity with it in the past and because we felt that the development community was a little larger than the other two databases we

considered, mSQL and PostgreSQL. Any of the three would have been a suitable choice for our project.

Portability

RPM and cfengine had already been reported to run on every OS on which we would need to run them. To assure portability, all we needed to do was to make sure that our cfengine modules were written in a language that was portable across all of our systems and had the ability to communicate with a MySQL server.

We made a list of all the scripting and programming languages we could think of with MySQL support and came up with the following: Perl, Python, Tcl, C, C++ and Java.

Of those, we immediately ruled out three choices. We decided not to write our modules in Python out of personal preference. We ruled out Tcl because the MySQL support was not as strong as we would have liked. We also ruled out Java because a Java virtual machine was not available for every operating system we would need to run it on.

With Python, Tcl, and Java gone, we were left to choose either Perl, C, or C++ which prompted an interesting question: were there any specific benefits or hindrances that made scripting languages a better choice than compiled languages (or vice versa)?

While personal preference was bound to affect our decision, we tried to be as objective as possible. These modules would not be doing large amounts of heavy computation, so any speed increase given by choosing a compiled language was likely to be negligible. These modules would be distributed from our central server through cfengine, which would mean very little difference between distributing one Perl script to all OSes or distributing a binary for each OS built from the same source code.

If we wrote our modules in Perl, we would need to make sure that every computer had a copy of Perl and a copy of the MySQL module for Perl. Whereas if we used C or C++, no additional software would need to be installed on the client machines; all we would need is a copy of the MySQL libraries and header files on one system running each OS in order to compile the C or C++ modules.

However, if we used Perl, it would be highly unlikely that we would ever use any of its more complicated features that might cause our scripts to react differently depending on what operating system they were being run on. As a result, if our scripts compiled and ran on one OS, they would likely compile and run on all the others. We wouldn't need to recompile our modules on eight different OSes every time we made a small change to either of our modules. Releasing new versions of the modules would also be easier because we wouldn't have to compile and test the program on all of our OSes with every minor change.

We eventually chose Perl because of the ease of development and maintenance it would give us over C

or C++. Every computer we would be responsible for maintaining would have a copy of Perl on it anyway, and it wasn't much trouble to add the MySQL module to it. It's not clear that this was the best choice, and in the future, we may rewrite our modules in C or C++ as an experiment.

Autonomous Operation

Building our system through the use of cfengine modules was the perfect solution for us. Because of the way cfengine handles class definitions through its modules, any of cfengine's internal functions could be performed on our systems based on what patches or software packages we installed – all automatically! All that is required is that our modules define classes for the patches and packages that they install; we can then have cfengine run commands based on these class definitions.

The possibilities are almost endless. We can have cfengine restart daemons after installing OS patches that update them. Cfengine can change `inetd.conf` and send `inetd` a hangup signal if we install or uninstall a software package that is launched from `inetd`. Nearly any change that you would want to make to a computer due to the installation or uninstallation of a software package or OS patch can be performed on all of the applicable hosts by merely adding a couple of lines to cfengine's input file.

Below, we show an example of a cfengine input file which illustrates how classes can be used by cfengine to reconfigure `inetd` based on whether or not the IMAP server package is installed.

```
editfiles:
  imapd::
  { /etc/inet/inetd.conf
    AppendIfNoSuchLine "imap stream tcp
      nowait root /usr/sbin/tcp imapd"
    DefineClasses "inetd"
  }

!imapd::
  { /etc/inet/inetd.conf
    DeleteLinesStarting "imap"
    DefineClasses "inetd"
  }

[later in the input file]

processes:
  inetd::
  "inetd -s"
  action=signal
  signal=hup
```

In this example, we show the use of two builtin cfengine commands, `editfiles:` and `processes:`, together with a dynamically defined class called `imapd::`. The `editfiles:` command provides a number of actions for manipulating text files, while the `processes:` command provides actions for manipulating UNIX processes. Through a process detailed in the next section, our software module will dynamically define the `imapd::`

class based on whether or not the IMAP server package is installed; if the package is installed, the class will be defined, and if the package is not installed, the class will not be defined.

Looking at the example above, you can see that if the `imapd` package is installed (i.e., the `imapd::` class is defined), `cfengine` will check to make sure that the `imap` service is enabled in the `inetd.conf` file. If a line for `imap` doesn't appear in `inetd.conf`, that means that the package has just been installed, at which point `cfengine` will append the line given in double quotes to `inetd.conf` and then define the `inetd::` class (this class will be defined if and only if `cfengine` edits the `inetd.conf` file as a result of this check). Later on in the `cfengine` run when the `processes:` command runs, `cfengine` will see that the `inetd::` class is defined and send the `inetd` daemon a HUP signal as instructed by the input file.

If the `imapd` package is not installed, then the `imapd::` class will not be defined by our module, and so `cfengine` will run the actions in the `!imapd::` stanza. These actions tell `cfengine` to remove the `imap` entry from `inetd.conf` if present and set the `inetd::` class if the entry was removed. If the `imap` line is present, that means that the package has just been uninstalled. In this case, `cfengine` will remove the `imap` line from `inetd.conf` and define the `inetd::` class, which will later, when the `processes:` command is run, cause `cfengine` to send a HUP signal to `inetd`.

Similar actions can be taken with the installation or removal of any software package or OS patch.

Summing It Up

We now had a good idea of exactly how we wanted to implement our system. It would be driven by `cfengine` so that every machine could be configured centrally from the `cfengine` server. We would have a MySQL database holding up-to-date patch information for each of our OSes and software configuration profiles for each of our hosts. We would need to write two `cfengine` modules in Perl: one to make sure that the computer was up-to-date on patches, and one to make sure the computer had the proper third party software installed.

Putting It Together (The Hard Part)

Cfengine and RPM

The first step in making this system materialize was to build a `cfengine` server and familiarize ourselves with `cfengine` in operation. For our `cfengine` server, security and economy were our top priorities, so we bought a Pentium-based system and installed OpenBSD on it. We disabled every service except for `ssh` and `ftp`, and loaded a highly restrictive `ipfilter` ruleset. We downloaded and installed `cfengine` and set up `cfengine`'s daemon to share input files with all of the systems on our subnets. We picked a handful of computers on which to install `cfengine`. These computers would contact our `cfengine` server every hour to

check for updated input files, download them if necessary, and run the commands in the input files. After two or three weeks of leisurely experimenting, we felt we were familiar enough with `cfengine` to begin designing a software and patch module system around it.

Since a large fraction of the computers we're responsible for are Sun SPARCs running Solaris, we started developing on our Solaris machines first. After making and installing a Solaris `pkg` package for RPM, we thought of all the third party software that we would want to install, downloaded the source code, and compiled them into RPM packages. This took about two months and was by far the most time consuming step of the entire process.

Software and Patch Database

Once we made all of the RPMs, the next step was to design and create a database to store all of our software information and patch information. Towards this end, we had to get into the specifics of exactly what kind of information we would want our database to hold. For software, we came up with the following list:

- The name of every package available.
- A listing of what OS each package was available for. Some of our software would only need to be compiled for one or two of our operating systems. For example, our Red Hat Linux systems would already have installed out of the box several packages which we would be building for our other systems. We needed to be able to differentiate, for example, between the "screen" package that shipped with Red Hat and the one that we built from our own spec file for OSes that didn't ship with a copy of screen. It would be very nice for our systems to be aware of which packages are installable and which ones not to bother with.
- The current, default version of each software package, applicable to every operating system.
- The ability to specify different current versions of the software packages for different operating systems. Some versions of a few of the software packages we'd be using simply wouldn't be runnable on some of our operating systems. It'd be great to say, "Everybody run version 1.24 of the 'foo' package, except for you AIX machines; I want you running version 1.22."
- A list of every software package that should be installed on every machine.
- A field in each machine's profile that would allow you to specify a specific version of a software package if you wanted a version other than the current. Some of our users are running software on their computers that require specific versions of packages like Perl or Tcl. It would be necessary to keep these packages from being replaced when we released new versions and marked them as being current.

Package Name	Default Version	Solaris Version	Linux Version	Solaris Build	Linux Build
acoread	4.05-1	NULL	NULL	1	1
tcl	8.1.1-1	NULL	NULL	1	NULL
pam_opie	1.1-1	1.0-2	0.21-3	1	1

Figure 1: Database table holding software information.

After we had come up with that list, we decided to use two database tables to hold this information.

One table holds information for each software package we want to distribute (see Figure 1). Each package has an entry in this table. One column in the table specifies the default version number of each package; this is the version of the package that is applicable to all OSes. Additional columns specify the version number of each package for each OS (Solaris Version, Linux Version, etc.). This version overrides the default version of a package for the given OS. If the value in the OS version column is NULL, then that means that the current version of the package for that OS is given by the value in the default version column. Finally, there are columns which are used to flag the availability of each package for each OS (Solaris Build, Linux Build, etc.). For each package, if the value in these columns is non-NULL for a given OS, then that means that we have built an RPM of that package for that OS. This can be used to distinguish between RPMs which we have built and those that are included with the OS.

For example, looking at the entries given in Figure 1, we can see that an acoread RPM is available for both Solaris and Linux, and that the default version of the acoread package is 4.05-1 for all OSes. On the other hand, a tcl RPM is isn't available for Linux (it wasn't necessary to make one, since Linux already includes tcl) but is for Solaris. Finally, we see that the default version of the pam_opie package is 1.1-1, but that Solaris systems should use version 1.0-2 while Linux systems should use version 0.21-3.

The second table holds information specifying which software packages should be installed on which machines (see Figure 2). For each entry in this table, one column specifies the hostname of the machine for which the entry applies, a second column specifies the name of the package that should be installed on that machine, and a third column specifies which version of the package should be installed on that machine. If the version is listed as "current," then that means that the current version of that package as given in the software table should be used. The collection of all entries for a machine in this table makes up that machine's software profile.

To illustrate, refer to the sample entries in Figure 2. From here we can see that the machine called

"mypc" (which is running Linux) should have the current version of acoread installed. As we saw before in Figure 1, the current available version of acoread for Linux is 4.05-1, so that means mypc will have acoread version 4.05-1 installed. Similarly, we can see that this machine will have pam_opie version 0.21-3 installed. Finally, we can see that machine "mysun" (which is running Solaris) will have tcl version 7.6-1 installed, even though the current version of tcl in the software table is 8.1.1-1.

Hostname	Package Name	Package Version
mypc	acoread	current
mypc	pam_opie	current
mysun	tcl	7.6-1

Figure 2: Database table holding host software profiles.

Patch No.	OS	OS Version
107451-04	solaris	5.7
108528-02	solaris	5.8
lpr-0.50-5	linux	6.2

Figure 3: Database table holding patch information.

The list we came up with for the patch database table was much simpler – the only thing we needed was a simple list of every patch for every version of every operating system we'd be maintaining. The table we used for the patch database is illustrated in Figure 3.

Once the database was designed and created, we needed to populated it with our software and patch

information. Since we would be the ones making all of the new software packages and deciding what software gets installed on what hosts, it would be easy for us to keep the software database up to date. Every time we released a new package, or wanted to install a piece of software on a host, we would just update the software database tables, either directly through the MySQL command line client, or through a front end (web-based or otherwise).

Keeping the patch database up to date would be a trickier affair. We would have no control over when new OS patches were released, and very few vendors or development teams would send us email whenever new patches were released. We would have to take the initiative of regularly going to each OS's patch information center, downloading any new patches, putting them up on our ftp server, and updating the database so our clients could pull them in.

To be clear, we didn't think going out to every OS's patch information site to check for new patches every morning would be a bad habit to get into, we just thought that we could find better things to do with the time that it would take to do that.

We decided to automate that process and run scripts from our cfengine server to go out and do it for us. Since these scripts would only be run from our central server, we could take existing software (such as PatchReport) and modify it to download new patches to our FTP server and optionally update our database. For OSes that we could not find any such software for, it was easy enough to write quick Perl scripts to check that vendor's WWW or FTP site, download new patches, and update the entries in our database. This way, we could check much more often than we could if we were doing it ourselves. Currently, we check every OS's patch information site every four hours, around the clock. Whenever the script downloads a patch, it emails us a notice containing a one-line description of each downloaded patch.

We should probably mention one precaution, however. In general, it is probably *not* a good idea to install a patch on a system without first examining any documentation that comes with it and trying it out on a test system first. The documentation will mention any special steps which need to be taken before or after the patch is applied, and prudent testing will uncover any potential problems that may be caused by a malformed patch.

In our current system, our patch download script automatically updates our database each time it downloads a new patch from the vendor. This means that our client systems will pull in and install the downloaded patches without our intervention. Given that we are only checking for "recommended" and "security" patches (the two patch types that seem to be well tested by the vendor before release), we aren't overly concerned about any potential damage resulting from automatic updates. We may revisit the wisdom of that

decision. If we do, all that will be necessary will be a quick change to disable the portion of the download script which updates the database. We could then just manually update the database after we have inspected and tested a patch.

Software and Patch Modules

Once we had the cfengine server and the software and patch database infrastructure built, all we needed was the client software.

From this point, it was relatively easy to write a cfengine module in Perl to grab a host's software profile, download the RPMs and install them. Since RPM was chosen to be the common package format across all of our OSes, no changes had to be made to this script in order for it to run on the different OSes. When run through cfengine on a system, this module will:

- query the MySQL database to generate a list of all software packages and their versions which should be installed on the system,
- generate a list of all software packages and their versions which are currently installed on the system (by locally running "rpm -q -a"),
- compare the two lists and install, delete and/or upgrade any packages on the system as needed (by locally running rpm),
- define a cfengine class for each software package that remains installed on the system.

After the module completes, cfengine continues its run and can act on any class definitions activated by the module. In this way, cfengine can make any configuration file edits, file copies, process signaling, etc. which are necessary as a result of any software changes, as illustrated with the imapsd example in the previous section.

The patch module was more tricky to write. Since all of our OSes handled patches in completely different ways, it was a challenge to make one script that would seamlessly patch as many as eight different OSes. It would be unavoidable to have certain parts of the script dedicated to only one OS, but we wanted to maximize the amount of code that would be common to all the OSes.

As with the software module, the patch module operates by first querying the MySQL database to generate a list of patches that should be applied to the current OS. It then compares this list to a list of patches currently installed on the system. Finally, it downloads and installs or updates patches as needed, using the native patch installation command for the OS. The module defines a class for each patch that it installs for further use during the cfengine run. For example, 107451-xx is a patch for the cron daemon in Solaris 7 for SPARC. Whenever a version of this patch is installed by the patch module, it will define the class 107451 which we can then use in our cfengine input files to tell cfengine to restart the cron daemon.

Some patches may fail to install on some systems. For example, a patch may be for an operating system component which isn't installed on the system (e.g., UUCP), or it may be intended for a specific type of system and not others (e.g., 64-bit systems and not 32-bit systems). While we could use the MySQL database to specify the applicability of a patch to a given system, we decided to only track the OS and the OS version for which a patch is applicable. As a result, our patch module may try to install inappropriate patches on a given system.

Fortunately, at least on the OSes for which we have so far implemented our patch system, the native patch installation tools which our module uses are intelligent enough to determine on their own that a given patch isn't applicable to the system and fail gracefully without damaging the installed OS. As a result, we are comfortable with our module trying to install inappropriate patches. Should the native patch installation tools not have the requisite intelligence, we will have to expand our patch database structure and module to accommodate. Fortunately, our design is flexible enough that this shouldn't be too difficult.

Given that some patches may legitimately fail to install, the patch module tracks failed patch installations using a text file on each system. When building its list of patches to install during a cfengine run, it will remove from the list any patches listed in this file; that way, it won't needlessly try to reinstall previously failed patches each time cfengine runs.

As a final step, both cfengine modules email us a report of their actions so that we can be kept aware of what software and patches are being installed on our computers. The report details which software packages and patches have been installed and/or removed, and includes any failures, errors or unusual output generated by the modules.

Summing It Up

We had now set up our cfengine server and our MySQL database, made RPMs of all our software, written the software and patch modules, and written the scripts to keep our patch information current. It was at last time to roll it all out!

Making It Work (The Fun Part)

Before we had started any work on this system, we agreed that every computer that was to be integrated into our management system would have to have its OS completely wiped clean and the current version of its OS installed. We wanted to minimize the number of versions of any given OS that were being used in our department, and since security was one of our primary motivations for undertaking this project, we also wanted to be assured that the OSes were clean and untruncated.

We would start by integrating all of our Solaris systems, then our Red Hat Linux systems, and then our IRIX systems. That would cover about 95% of all

of our computers, after which we would declare victory and integrate the other OSes as time permitted.

We picked two Solaris machines with particularly understanding users to use as guinea pigs, completed their software profiles, backed up their home and data directories, and marched off to install Solaris 7.

Once we had a clean installation of Solaris, we loaded on the RPM package (in Solaris pkg format), installed the Perl RPM with MySQL support, and the cfengine RPM. Sun's JumpStart program made this process extremely easy to do, all completely automated. We ran cfengine for the very first time, expecting the worst, and waited for the email report.

Much to our surprise, it worked flawlessly. All of the configuration files were modified just the way we had laid out in cfengine's input files, all of the patches we wanted applied were applied, and all of the software we put in the database configuration profiles was properly installed. After inspecting the systems for anything that was out of place, we declared the debut of our system a smashing success and made plans to upgrade the rest of our Solaris systems. At the time this paper is being written, all of our Solaris systems have been upgraded, and we're about halfway through our Linux systems. Upgrading and integrating the Linux systems has proven to be equally painless.

So far, our system has not caused any serious problems, and we've repelled numerous attacks on vulnerable services because our systems are never more than a few hours behind on vendor-supplied patches.

Successes

In this section, we figured it would be appropriate to include a few examples of where our system has clearly shown itself to be more useful to use than our previous hodge-podge method of systems administration.

Breakin Detection and Termination

Cfengine gives you an excellent interface to interact with your system's processes. You can search for specific processes, start them if they're not running, send them signals if they are, and email you with a report of what cfengine has found or done.

Since the people who break into our systems almost exclusively use the compromised systems to run sniffers, IRC bots, or DoS tools, we decided to make up a list of suspicious process names to have cfengine look for and warn us about every time it ran. Besides the usual suspects (more than one running copy of `inetd`, anything with "sniff", "r00t", "eggdrop", etc. in the process name, password crackers, etc.), we had cfengine watch for any process with "/" in the process name.

One afternoon, we got an email from cfengine on one of our computers that had noticed that the regular

user of that machine was running a program as `"/irc"`. It wasn't uncommon to see our users using `"/"` to run programs, nor do we have objections to our users running IRC, but in this case, it was a bit unusual for this particular user to be running an irc process (good UNIX system administration practice also dictates that you know your users).

Poking around the system, we discovered that the person running this program was not the regular user of the machine, but was someone who had evidently sniffed our user's password from somewhere else and remotely logged into his system just minutes before cfengine had alerted us. This person was in the process of setting up an IRC bot and had not yet tried to get a root shell.

At this point, we had to figure out exactly what to do to minimize the amount of damage that this person could cause with our user's password. We were about 40 minutes from the next cfengine run, so we had 35 minutes to make changes to cfengine's input files so that they would be pulled in by all our clients. The machine that this person had broken into is part of a group of computers in a specific professor's research group. Every user who works with this professor has accounts on every one of his machines, and almost all of the users use the same password on all of the machines. That meant that the person who broke into the one machine we had noticed had a password that would likely let him break into about 10 other machines. It was very important to us that we block them from being able to get into all these other machines. It would only be a matter of time before they figured it out and began installing programs all over this professor's research workstations.

Through cfengine, we had set up syslog on all of our clients to log all messages sent to the AUTH facility to a central log server. Based on the logs on the server, we could say with relative certainty that this person had only managed to break into the one system. We also had the IP address that he had come from. We decided that we would boot the cracker out of the compromised account, lock out our user from all of the machines on which he had accounts, and block all IP access to every machine in this researcher's group from the entire class C from which the attacker had come.

In our Solaris cfengine input file, we added a rule for every computer in that professor's group to look in `/etc/shadow` for the regular expression `"user:.*:"` and replace it with `"user:LK*:"` (we don't use NIS or an equivalent, so each system has its own local passwd and shadow file). We also made a change in the global ipfilter ruleset to drop and log every packet coming from the class C from which the breakin had come.

Just before the next cfengine run took place, we killed off the intruder's shell. Two minutes later, the user had been locked out of all of the research group's

machines until we could get in touch with him and get a new password. The sniffed password the cracker had obtained was now completely useless, and he had lost all access to our machines from the host that he was using as a breakin staging point. We watched the logs and systems intently for the next few days and saw absolutely no sign that this person was trying to get in again.

All within the time frame of one hour, we had detected the breakin, determined the cause, terminated it, and prevented it from happening again. With the old way of doing things, it's doubtful we would have ever caught this person unless he had launched a DoS and caught the attention of our campus-wide network security team. Score one for the good guys!

Breakins Prevented Through Proper Patching

Far less dramatic, but just as satisfying, was a recent experience with a new Linux bug.

In the old way of doing things, we would be relatively free of breakins (as far as we knew) for periods of time, and then some script kiddie would find an exploit and decide to go to town on the university. We've had days that began with the campus network security team giving us a list of 20 machines on our local subnets that they'd seen compromised and had pulled off the network. We would have to drop everything that we were working on for two days to clean up these systems. These breakins would occasionally be for newly found vulnerabilities, but more often would be for vulnerabilities that had been around long enough to have had patches released. We'd get hit because we hadn't had enough time to patch any of our computers against the vulnerability.

Recently, we've seen several attacks against a bug in `rpc.statd` on campus. When they first started to show up, we braced ourselves for the worst, but after comparing the CERT advisory that addressed this vulnerability (CERT Advisory CA-2000-17) against Red Hat's errata page and the patches we had installed, we found that our system had already downloaded and installed an updated version of `rpc.statd` that was invulnerable to this specific attack. We had been safely patched for weeks.

Conclusions

So far, our system has proven to be well worth the effort it took to bring it to fruition. We rarely have to worry about system vulnerabilities because all of our systems automatically patch themselves as soon as new patches are released. We update software regularly now because we can easily compile it once, and install it on dozens of systems simply by editing one entry in one database table. We have regained control of our network.

Availability

All of the scripts that we're using to run our system, including the patch module, software module,

and the CGI script we use to configure the software profiles for all of our hosts are freely available at: <http://astro.uchicago.edu/~davidr/cfengine-tools/>.

Acknowledgments

Our system has proven itself useful in more ways than we ever could have expected. A great deal of credit goes to the author of cfengine, Mark Burgess. Through the use of his wonderful software, we've been able to bring order to our systems that we would have never had been able to otherwise. We would also like to extend a special thanks to our LISA shepherd, David Blank-Edelman. Without his seemingly endless patience when it came down to the last few days, our paper would not have been nearly as good as we feel it is now.

About the Authors

John Valdés has been playing with UNIX since 1986 and has been working as a System Administrator since 1990. He currently manages systems for the Department of Astronomy and Astrophysics at the University of Chicago. John can be reached via email at valdes@uchicago.edu.

David Ressman has been a UNIX System Administrator since 1996. He currently works for John in the Department of Astronomy and Astrophysics at the University of Chicago. He enjoys referring to himself in the third person, and he hopes to begin college in the Fall of 2001. David can be reached via email at davidr@oddjob.uchicago.edu.

References

- [Abb97] Abbey, Jonathan, *The opt depot Web Site*, http://www.arlut.utexas.edu/csd/opt_depot/opt_depot.html.
- [Bai97] Bailey, Ed, *Maximum RPM, Taking the Red Hat Package Manager to the Limit*, August 1997, Macmillan Computer Publishing.
- [Bur95] Burgess, Mark, "Cfengine: a site configuration engine", *USENIX Computing Systems*, Vol 8, No. 3 1995.
- [Bur99] Burgess, Mark, *Cfengine Reference Manual*, <http://www.iu.hioslo.no/cfengine/docs/cfengine-Reference.html>.
- [Col92] Colyer, Wallace and Walter Wong, "Depot: a Tool for Managing Software Environments", *Proceedings of the 6th Systems Administration Conference (LISA VI)*, 1992.
- [Gli96] Glickstein, Bob, *The GNU Stow Web Site*, <http://www.gnu.org/software/stow/stow.html>.
- [MySQL] MySQL, <http://www.mysql.com/>.
- [Oet98] "SEPP – Software Installation and Sharing System", Tobias Oetiker LISA 1998.
- [Rou94] Rouillard, John P. and Richard B. Martin, "Depot-Lite: A Mechanism for Managing Software", *Proceedings of the 8th Systems Administration Conference (LISA VIII)*, 1994.
- [RPM00] *The Red Hat Package Manager Website*, <http://www.rpm.org/>.
- [Sel91] Sellens, John, "Software Maintenance in a Campus Environment: The Xhier Approach", *Proceedings of the 5th Large Installation Systems Administration Conference (LISA V)*, 1991.
- [Sin97] Singer, Daniel E., "ToolMan Meets PatchReport", *login: The magazine of Usenix and SAGE*, October 1997.
- [Yar99] Yarger, Randy Jay, George Reese & Tim King, *MySQL and mSQL*, July 1999, O'Reilly and Associates.

Unleashing the Power of JumpStart: A New Technique for Disaster Recovery, Cloning, or Snapshotting a Solaris System

Lee "Leonardo" Amatangelo – Collective Technologies

ABSTRACT

Due to the demand of 24x7 coverage by present day data centers, the need for a proven disaster recovery plan is a must. To assist in providing disaster recovery for systems running Sun Microsystems' Solaris 2.x (SunOS 5.x) operating system, a tool was developed which captures the image of the system to one or more volumes of optical media with the first volume being bootable. The optical media used by this tool is CD (Compact Disc) with hooks put in place for DVD (Digital Video Disc or Digital Versatile Disc).

This tool was developed with the following objectives in mind: (1) no magnetic media; (2) bootable media; (3) minimal user interaction; (4) handle multiple volume sets; (5) handle environments that do not use a Network Information Service (NIS or NIS+); and, (6) handle environments that do not use Network File System (NFS). The power of this tool is made possible by utilizing two special features present in the Solaris operating system, namely the installboot utility and the JumpStart mechanism (as implemented on the Solaris 2.x Install CD).

The complete process of capturing and restoring a Solaris system's image to bootable optical media involves five phases: (1) pre-imaging preparation; (2) setup of target host; (3) capture image of target host; (4) burn image to media; and, (5) restore image to target host. Each of these phases is controlled by one master Bourne shell script. The overall tool is implemented in 26 Bourne shell scripts controlled by the 5 master scripts. This tool, known as the "CART" (Capture And Recovery Tool), was placed on a mobile cart and consisted of the following components: UltraSparc 10, internal CD-ROM drive, internal floppy drive, 256 MB memory, 2 network interface cards (NIC), keyboard, mouse, monitor, external SCSI 18 GB disk drive, external SCSI CD-RW, and external SCSI DLT.

This tool provides the following functions: (1) bare-metal recovery; (2) capture a snapshot of a system to optical media; (3) cloning a system; and, (4) rollout multiple clones of a system via optical media. Using the principles discussed in this paper and creating additional Bourne shell scripts, the CART has been modified to provide the following additional outputs: (1) make copies of the Solaris Install CD; (2) make customized Solaris Install CD (essentially a customized JumpStart from CD); and, (3) a specialized bootable CD for disaster recovery that assists third party Backup and Recovery utilities (Legato Networker and Veritas NetBackup).

Introduction

With the increase of e-commerce on the Internet, there is an ever-increasing demand to have mission critical computer systems run 24 hours by 7 days for long periods between scheduled down times. When a mission critical system suffers from corrupted disk data, either software or hardware induced, the disk data will need to be restored. Software induced data corruption include application malfunctions that render data inaccessible or a user accidentally entering a data destructive keystroke such as `rm -r`. Hardware induced data corruption include physical disk crashes. In the case of physically damaged disk drives, the disk drive of course will need to be replaced prior to the restoration of software and data.

When the disk drive in question happens to be the boot drive, the situation becomes more involved.

One cannot simply run the backup application to restore the system because there is no longer a viable operating system running (nor present) on the system. Under such dire circumstances, rebuilding a boot disk can take what seems like an inordinate amount of time. The operating system needs to be installed, standard packages (clusters) installed, customized packages installed, patches applied, kernel tuning, and finally any special or custom configurations need to be set. To perform all of these tasks could take up to a couple hours. Even using Sun Microsystems' automated install utility, JumpStart, can take what might be considered too much time, because following the installation of the operating system and any customized packages, the Jumpstart process will still need to initiate the installation of the desired patch sets. The application of a large set of patches can take up to a couple of hours.

Due to the lengthy time it sometimes takes to patch a system, Hewlett-Packard's automated install utility for the HP-UX operating system, "Ignite-UX," can prove to be more efficient time-wise to install a system. When Ignite-UX is finished, the system will be complete with operating system, applications, patches, kernel modifications, and any specialized configurations. Ignite-UX uses the concept of saving an entire system image of a benchmark host, known as a "golden image," and then restores the image during the automated rollout. The Ignite-UX process essentially performs a bit-by-bit copy, and therefore, is much faster than the JumpStart process, which builds the target system from the ground up by installing the operating system, software packages, patches, and any additional applications and files desired.

Taking the best of both the JumpStart and Ignite-UX worlds, a tool for providing disaster recovery, cloning, or snapshotting has been created specifically for Solaris systems.

The Capture And Restore Tool – The "CART"

During its evolution, this tool passed through a client site that needed to maintain the utmost in security. As such, the client site did not run network services to ease the administration nightmare of maintaining the numerous users, hosts, and system administrative files (usually kept in the /etc directory). Thus, this client did not use Network Information Services, NIS, nor its more secure brother, NIS+. This client also did not use the Network File System, NFS, to share resources over the network for fear that such a service was not secure enough for the very confidential information that was stored on the network. In order to accommodate this client, the tool was made to be self-contained and lived on a mobile cart so that the tool could be easily negotiated through the data center. The system hosting the tool will be referred to as the "control host" while the system to be imaged will be referred to as the "target host." The two hosts will talk to each other via a directly connected network cable.

This mobile tool physically consists of the following components: UltraSparc 10, internal CD-ROM drive, internal floppy drive, 256 MB memory, 2 network interface cards (NIC), keyboard, mouse, monitor, external SCSI 18 GB disk drive, external SCSI CD-RW, and external SCSI DLT. Additionally, the system has a network cable attached to one of the NICs and an RS-232 serial cable attached to serial Port B. The RS-232 cable was a convenience added to handle headless hosts via the 'tip' utility.

The main purpose of the tool discussed in this paper is to capture and restore system images. Furthermore, the tool physically resides on a mobile cart. Because of these two reasons, the tool became known as, and will be referred to during the rest of this paper, as the "Capture And Restore Tool," or simply the

CART. Figure 1 diagrammatically depicts the physical layout of the CART.

Two Special Solaris Features

The secret and power of the CART is brought about by two very special features found in the Solaris 2.x operating system. These two features are specifically the *installboot* utility and the JumpStart mechanism. The former facilitates the placing of a bootblock on storage media while the latter facilitates a way to customize the resultant boot process. These two features are further explored below.

How to Make Solaris Storage Media Bootable

One of the objectives of the CART was to restore a system via bootable optical media. The first question to ask is, "Is it possible to make bootable media under the Solaris 2.x operating system?" The answer is a resounding YES! The solution is to use the *installboot(1M)* utility. The *installboot* utility, uniquely found in the UNIX command sets distributed in Sun Microsystems' Solaris 1.x (SunOS 4.x) and Solaris 2.x (SunOS 5.x) operating systems, provides the capability to install a bootblock on various types of storage media. Furthermore, the bootblock can be placed on any of the slices (partitions) on the target media.

The second question to ask is, "Can a bootblock be placed on optical media?" Again, the answer is a resounding YES! As a matter of fact, the bootblock can be placed on various types of bootable media, such as a floppy diskette, hard disk drive, CD-R, CD-RW, CD-ROM, DVD-R, DVD-RW, DVD+RW, DVD-ROM, and DVD-RAM. When the bootblock program resides in the boot area of a disk partition, the bootblock program will load the *boot(1M)* program, also known as *ufsboot*. The *ufs* boot objects are platform-dependent, and reside in the */usr/platform/`uname -i`/lib/fs/ufs* directory, where 'uname -i' gets the correct platform-name. The full command syntax for *installboot* is as follows:

```
installboot bootblock raw-disk-device
```

where:

bootblock is the name of the bootblock code.

raw-disk-device is the name of the disk device onto which the bootblock code is to be installed; it must be a character device which is readable and writable. Naming conventions for a SCSI or IPI drive are of the format: *c?t?d?s?*. Naming conventions for an IDE drive are of the format: *c?d?s?*.

For a thorough explanation of the *installboot* utility, see the Man Pages on *installboot(1M)*. In short, the *installboot* utility was used to install the necessary bootblocks so that the CART could handle the various Sun Microsystems platform architectures. More on how this was accomplished later.

A Brief Discussion on JumpStart

There are plenty of books on the subject of JumpStart [1, 3, 6], and as such, the scope of this paper is

not to teach JumpStart. However, a brief discussion of JumpStart will help elucidate its role and importance as implemented in the CART. The JumpStart feature was introduced by Sun Microsystems and SunSoft in 1992 and first appeared in the Solaris 2.x operating system [3]. The JumpStart feature is primarily used to install multiple client hosts over the network simultaneously with the intent to lessen this tedious and time-consuming task. The various hosts can have their own unique configuration if desired or groups of hosts can have like configurations. JumpStart is a utility to help facilitate the ease of rolling out the Solaris operating system to new hosts (or existent hosts to be upgraded) in an environment and is especially useful for rolling out a large number of installs across an enterprise [2, 5].

Typically, JumpStart is used over a network to allow the automated software installation rollout of multiple hosts simultaneously. Once configured, the JumpStart mechanism is very much hands off. To facilitate the JumpStart mechanism over a network, a networked JumpStart server is needed. But JumpStart servers are not the only place that one will find the implementation of JumpStart. Sun Microsystems has also incorporated the JumpStart mechanism on the bootable installation media (i.e., the Solaris 2.x Install CD) as the means for installing the Solaris operating system.

The JumpStart mechanism has four main parts: (1) a BEGIN script which performs pre-processing actions to take place prior to the software installation; (2) a PROFILE which defines how Solaris is to be installed on a particular client; (3) a FINISH script which performs any post-processing actions following

the software installation; and, (4) a rules file (rules.ok) that dictates which BEGIN script, PROFILE, and FINISH script to use.

Because JumpStart offers the following three characteristics: (1) can be configured to have minimal user interaction; (2) can be placed on optical media; and, (3) can be invoked seamlessly following a boot, JumpStart was selected hands down as the method of choice for the CART.

As will be described in the next section, the CART ended up customizing JumpStart in a completely different way and was able to unleash the true power of JumpStart!

The "CART" in a Nutshell

To make use of the existent JumpStart technology as implemented on the Solaris Install CD, the CART performs the following steps: (1) partitions a hard disk drive (which will be referred to as the "image disk") to have slices that mimic (size and location) the Solaris Install CD; (2) creates filesystems on the slices; (3) copies select contents of the Solaris Install CD to the corresponding slices on the "image disk"; (4) opens up the permissions on the directory and files where the JumpStart mechanism resides; (5) replaces the standard issue JumpStart BEGIN script, PROFILE, FINISH script, and RULES.OK file with its own highly customized versions; and, (6) installs the appropriate platform-specific bootblock that corresponds to the platform of the "target host." See Figure 1 for a quick overview of CART's hardware configuration.

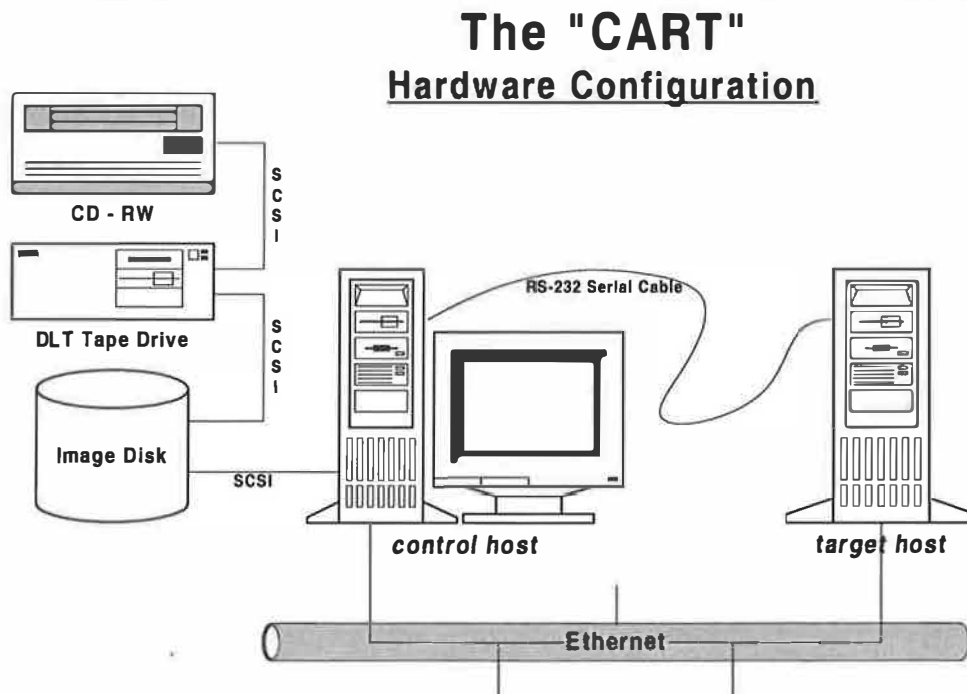


Figure 1: CART hardware configuration.

At this point, if the contents of the “image disk” is burned to optical media, that optical media will be bootable and will go through an automated install based on instructions in the customized JumpStart files, specifically the customized BEGIN script and the RULES.OK file. The customized BEGIN script ends with a system call to reboot the system. This reboot essentially causes the normal JumpStart process of loading packages, patches, etc. not to occur. In short, the CART makes use of the Solaris Install CD JumpStart mechanism only as a means to provide the automatic running of a script following the boot from optical media. It was decided not to reinvent the wheel but instead to modify an existent proven technology. The steps just described are what truly give the CART its power.

This technique can be used for producing a wide range of useful applications, some of which will be discussed later in this paper. The final implementation of the CART did not use a PROFILE or a FINISH script. However, a customized FINISH script could be implemented to initiate backup software to restore variable data from the latest backup of the system.

After the “image disk” has been prepared successfully, the next task of the CART is to capture the entire system image of the “target host.” To accomplish this task, a private network is setup between the “control host” and the “target host” with NFS mounts established between the “target host” and the “image disk” (located on the “control host”). Next, the “target host” is interrogated to determine its number of disk drives. The partition table of each disk drive is logged capturing the slices used and the sizes of those slices. The CART then methodically runs through each disk drive and captures an image of each slice. As the image for each slice is captured, the CART logs the size of the slice image to a file. The size of each slice is captured for two reasons. First, to determine if an individual slice image will be able to fit on a single optical media volume. This is important in the case when saving to CD because a single volume can only hold 650 MB. The second reason is to determine how many optical media volumes will be required to save the entire system image of the “target host.”

Once the entire system image of the “target host” is captured, which typically will be comprised of several individual slice images, the entire system image must be burned to the optical media of choice. The log containing the size information of the individual slice images is used to determine which slices get placed on which volume. The user of the tool will be informed how many volumes will be required to burn the entire system image and as each volume is burned the user is requested to remove the volume and insert the next volume.

Once all of the optical media volumes have been successfully burned, the result will be a snapshot of the entire “target system” captured onto a set of

optical media, with the first volume being bootable and having the smarts to recreate the “target host.”

The last step to complete this eventful saga is to perform the restore of the image from the bootable media, either to the original “target host” or to an identical host with as large or larger disk drives. The restore takes place at the Open Boot PROM (OBP) level by booting from the optical media.

The Five Phases of the “CART”

The capture and restore of a Solaris system image using the CART involves five distinct phases.

Phase 1, the pre-imaging preparation of “image disk” phase, prepares a buffer area (hard disk space) known as the “image disk.” The “image disk” will have appropriate slices (partitions) and filesystems created for holding select directories and files of the Solaris Install CD, as shown in Table 2. These select directories and files contain the JumpStart mechanism and the bootblocks for the various Sun Microsystems architectures. In addition, the “image disk” will have a separate slice to hold log files during the capture phase and another slice to receive the compressed filesystem images from the target host.

Phase 2, the setup of target host phase, establishes a private network connection and NFS mounts between the “target host” and the “control host” that has access to the “image disk.”

Phase 3, the capture image of target host phase, determines the number of disks on the target host, the number of partitions on each disk, and then proceeds to capture (and ufsdump) compress (compress or gzip) each filesystem found. All disks, partitions, and compressed image file sizes are logged during the process. These logs will be used during the following “burn” and “restore” phases.

Phase 4, the burn image to optical media phase, creates a bootable volume and any additional volumes required for holding the complete system image. If the optical media is CD-R and the entire set of compressed images from the target host exceeds 650 MB, then the appropriate number of additional CD-R volumes will be created. If the optical media is DVD-R and the entire set of compressed images from the target host exceeds 3.9 GB, then the appropriate number of additional DVD-R volumes will be created.

Phase 5, the restore image phase, involves restoring the system image to the same or equivalent target host via bootable media. The boot process of the restore proceeds through a highly customized JumpStart process located on the bootable media itself. If the entire system image is contained on more than one volume, then an additional CD drive or DVD drive

(depending on which optical media is being used) will need to be attached to the host receiving the image.

Commands for the Five "CART" Phases

The commands to invoke each of the five phases of the CART are listed below:

Phase 1: Pre-Imaging Preparation of "Image Disk". The command for this phase is entered on the "control host" and can be run prior to connecting the "control host" to the "target host":

```
prepare_image_disk
```

Phase 2: Setup of Target Host. The command for this phase is entered on the "target host" after installing the "setup" floppy diskette, CD, or DVD:

```
setup
```

Phase 3: Capture Image of Target Host. The command for this phase is forked on the "target host" by the Setup phase (if the setup process is successful):

```
capture_system_image
```

Phase 4: Burn Image to Optical Media. The command for this phase is entered on the "control host":

```
burn_system_image
```

Phase 5: Restore Image. The command entered at the open boot PROM 'ok' prompt on the original "target host" or a similar host that is to receive the image:

```
ok boot cdrom
```

or

```
ok boot dvd
```

(The second option currently being developed by Sun Microsystems as a standard option in the OBP.)

A Closer Look at the Solaris Install CD

The Solaris Install CD was reverse-engineered to determine how it boots and goes through its JumpStart process. The first area looked at was the logical layout of the Solaris Install CD itself. Table 1 reports this layout, specifically for the Solaris 2.6 Install CD.

The first volume in the set of media containing the system image of the target host will emulate the Solaris Install CD layout above with three minor modifications. First, "slice 0" is trimmed down to contain only the bare essentials to have the CD boot and run through the customized JumpStart process. Second, a seventh slice, "slice 6," is added to the CD. This slice will contain the log files generated during the image capture phase. Third, an eighth slice, "slice 7," is added to the CD to hold some (if not all) of the compressed image files from the "target host." The emulation of the Solaris Install CD with these

modifications is captured on the "image disk" and is reported in Table 2.

Slice	Partition	Contents	Size
0	a	Installation and Distribution	600 MB
1	b	Miniroot	40 MB
2	c	Boot Info, sun4c*	1 cyl.
3	d	Boot Info, sun4m*	1 cyl.
4	e	Boot Info, sun4d*	1 cyl.
5	f	Boot Info, sun4u*	1 cyl.

Table 1: Solaris Install CD Layout.

* Note that these slices contain the boot information (bootblock) for the various hardware architectures of Sun Microsystems' products that run Solaris. Also contained in these slices is the file `.SUNW-boot-redirect` which contains a single byte, the character '1', to direct the firmware boot PROM program to look for the kernel on "slice 1" of the boot device.

Slice	Part.	Contents	Size
0	a	Installation and Distribution	200 MB
1	b	Miniroot	40 MB
2	c	BootInfo - sun4c *	1 cyl
3	d	BootInfo - sun4m *	1 cyl
4	e	BootInfo - sun4d *	1 cyl
5	f	BootInfo - sun4u *	1 cyl
6	g	Log Files	11 MB
7	h	Compressed Image Files	Rem'dr

Table 2: Image Disk layout.

* The contents of these slices are as described in the note in Table 1.

Also note that the CART implemented an 18 GB disk drive for the "image disk." The size of the "image disk (or disks)" is determined by the size of disk space on the target host.

Location of the JumpStart files on the Solaris Install CD and the "CART"

During the "pre-imaging preparation" (Phase 1), select portions of the Solaris Install CD are copied to the "image disk." Two specific portions are copied: (1) all of "slice 1," which contains the miniroot (the operating system); and, (2) parts of "slice 0," which contains the pertinent files to allow the JumpStart mechanism to function. The location for these files is as follows:

```
/s0/Solaris_2.6/Tools/Boot/usr/
    sbin/install.d/install_config
```

The pertinent standard issue JumpStart files are the following:

```
rules.ok Install JumpStart "RULES" file
install_begin Install JumpStart "BEGIN" script;
    called out by the "rules.ok" file
devsyn_finish Install JumpStart "FINISH" script;
    called out by the "rules.ok" file
```

How the “CART” Customized JumpStart

Since, the JumpStart files (listed in the section above) now exist on the “image disk,” and the tool runs as ‘root’ user, the permissions on the location and the files themselves can be modified. Once the permissions are changed, the files can be modified (replaced) with highly customized versions, which is precisely the approach taken by the CART to highly customize the restore (Phase 5).

The customization involved replacing the “rules.ok” file and adding a custom JumpStart BEGIN script, named “restore_system_image_begin,” to the JumpStart location on the “image disk.” The CART did not use the standard issue JumpStart files “install_begin” and “devsyn_finish.” Furthermore, the CART did not need a custom JumpStart FINISH script at all. All the actions the CART needed to accomplish were performed in the custom BEGIN script “restore_system_image_begin.” These actions included verifying all of the appropriate disks are present, the disks then get partitioned, filesystems get created, and one by one, the filesystem images get restored. As additional volumes containing the system image are needed, the user is instructed to remove the current volume and insert the next. When all of the filesystems are restored and an appropriate bootblock has been installed, the customized BEGIN script reboots the system. When the “target host” finishes rebooting, it will look like the original host. The reboot at the end of the customized BEGIN script also has another side effect, it aborts the rest of the JumpStart process. Thus, the system does not try to load the distribution, packages, nor patches. Instead, the completed process looks more like an Ignite-UX “golden image” restore.

Functional Details of the Five “CART” Phases

For those who want to know the nitty-gritty details of the entire CART process, the functional details for all five phases are itemized below.

Phase 1: Pre-Imaging Preparation of the “Image Disk”

1. Requests user to perform physical setup
2. Unmounts the “image disk”
3. Partitions the “image disk”
4. Creates filesystems
5. Makes mount points for mounting the “image disk”
6. Mounts the “image disk”
7. Places proper permissions on the “image disk”
8. Copies the contents of the “Solaris Install CD” to the “image disk”
9. Removes the “lost+found” directories from slices s0-s7
10. Installs the various architecture bootblocks
11. Opens the permissions on the JumpStart locations in slice s0 on the “image disk”

12. Creates additional directories with proper permissions in the miniroot filesystem (slice s1) of the “image disk”
13. Share the “image disk” and other local shares for the NFS mounts from the target host

Phase 2: Setup of Target Host

1. Creates the work directory on the target host
2. Saves the original /etc/hosts file
3. Reassigns a temporary IP Address
4. Determines the network interface type
5. Saves the original network interface settings
6. Creates a virtual network interface
7. Establishes a network connection between the target host and the master control host
8. Logs all changes so the original configuration can be re-established when complete
9. Forks the execution of the “capture image” script

Phase 3: Capture Image of Target Host

1. Determines the physical device name of the optical media (CD or DVD) drive
2. Saves /etc/vfstab file information
3. Determines the number of disk drives on the system using format
4. Saves the partition information of each disk drive using prtvtoc
5. Performs ufsdumps of the individual filesystems
6. Compresses the filesystem dump files
7. Calculates the number of media volumes that will be required to hold the entire image (compressed dump files) of the system
8. Places the appropriate bootblock on the “image disk”
9. Copies the customized JumpStart BEGIN script to the JumpStart location on slice s0 of the “image disk”
10. Copies the customized “rules.ok” file to the JumpStart location on slice s0 of the “image disk”
11. Restores the original /etc/hosts file
12. Restores the original network interface settings

Phase 4: Burn Image to Optical Media

1. Kills the volume management if it is running
2. Checks that the “image disk” is mounted
3. Reads the number of volumes to create from the log files
4. Puts files in ISO-9660 format
5. Creates (burns) the bootable first volume
6. Writes a label on the first volume indicating Volume 1 of ‘x’ number of volumes and target host name
7. Creates (burns) all additional volumes
8. Writes a label on each volume indicating Volume ‘y’ of ‘x’ number of volumes and target host name

Phase 5: Restore Image

1. The target host to receive the image is booted from the first volume of the optical media set

2. Customized JumpStart located on the first volume of the optical media automatically gets initiated
3. Mounts slice 6 of the optical media to /log_dir
4. Uses 'finthard' to recreate the VTOC of the original disks to the target host disks
5. Creates new filesystems on the appropriate slices of the new disks to match the filesystems from the original disks
6. Mounts one at a time the intended slices on the new disk to the mount point /mnt
7. Uncompresses and ufsrestores the compressed image files
8. Places the pertinent bootblock on the boot device

Handling the CD Mounts During Installation

This paper would be remiss if it did not describe the manner in which the CART mounts the bootable first volume upon a restore of the image (Phase 5). The mounting of the bootable CD is identical to how a Solaris Install CD gets mounted during an install. This information is important because it elucidates one of the limitations of the CART, that being that if the CART produces a multi-volume set during the burning of the media (Phase 3), then an additional CD drive will be required during the restore of the image (Phase 5).

The details on how the contents of the bootable CD get mounted at boot time, is shown in Table 3.

/tmp	/tmp
/proc	/proc
/devices	/tmp/devices
/dev	/tmp/dev
/	/devices/pci@1f,0/pci@1,1/ide@3/atapicd@2,0:b
/cdrom	/devices/pci@1f,0/pci@1,1/ide@3/atapicd@2,0:a
/dev/fd	fd

Table 3: The boot-time mounts of the CART bootable CD.

Note: The information in this table is for an Enterprise 250. The / and /cdrom mounts will vary depending upon the hardware platform being used.

Table 3 reports that the miniroot (slice b on the optical media) represented by the line ending with a "b," is being mounted to the / root mount point. The miniroot is the operating system being used for the restoration (in the case of the CART) or the installation (in the case of the Solaris Install CD). Since the operating system must always be present, the first volume must always be mounted during the entire restore (or install) process. Thus, a second media (CD or DVD) drive must be attached to the "target host" in order to restore a system image contained on a multi-volume set.

When booting from a CART generated bootable first volume or from a Solaris Install CD, the CD will get mounted as in Table 3.

Optical Media

The optical media used by the CART is CD and DVD.

CD technology is attractive because it is inexpensive. DVD technology is attractive because it can hold much larger capacities.

After speaking with several vendors [10, 11, 12, 14], the same message was echoed that there is really only one vendor, Pioneer, that provides a DVD writer. The vendors that provide CD/DVD recording solutions (hardware & software) use the DVD writer specifications from Pioneer.

In terms of the writable media, there is a dramatic increase in cost going from CD to DVD: ~\$1 per 650 MB CD vs. ~\$35 per 3.95 Billion Bytes DVD (which is not really 3.95 GB in the computer sense of the word but rather in the true mathematical sense of the word – which is a marketing ploy; see the note under the "DVD Technology Capacity and Compatibility" section below).

So far, Young Minds, Inc., (YMI), a leading CD recording technology vendor, is the only vendor I found working on a DVD solution for UNIX. In early 2000, YMI was awaiting hardware and specifications from Pioneer. Furthermore, the current CD solution from YMI is scalable to handle DVD; all that is required is a DVD writer and a PROM update to the YMI CD Studio hardware.

Unlike the CD technologies, where the CD-R, CD-ROM, and CD-RW versions all have the same capacity, each version of DVD technology has a different capacity. See tables below. The varying capacities of DVD technology will become a source of confusion for the consumer primarily because the various DVD technologies will not be compatible with one another. Currently, for non-video applications, the DVD technology market still needs to sort itself out. On the other hand, the CD technology market is well established.

To learn more about CD and DVD technology, turn to the web [7, 8].

CD Technology Capacity and Compatibility

Currently, there are four CD versions: CD-R, CD-RAM, CD-ROM, and CD-RW. All have the same capacity and are compatible with one another:

Type	Capacity
CD-R	650 MB
CD-RAM	650 MB
CD-ROM	650 MB
CD-RW	650 MB

DVD Technology Capacity and Compatibility

The DVD standards committee is still trying to decide on the DVD standards [7, 14]. Currently, there are five DVD versions: DVD-R, DVD-RAM, DVD-ROM, DVD-RW, and DVD+RW. These versions have varying capacities, and thus, are mostly incompatible with one another:

Type	Capacity
DVD-R	3.95-4.7 Billion bytes
DVD-RAM	2.6-4.7 Billion bytes
DVD-ROM	4.7 Billion bytes
DVD-RW	4.7 Billion bytes
DVD+RW	3.0-4.7 Billion bytes

As an aside, these capacities are truly in billions (1,000,000,000) of bytes and not a Gigabyte (GB) in the computer sense of the word, where a GB is defined as 1024 x 1024 x 1024 or 1,074,790,400 bytes.

Where Do We Go From Here

There are a couple of areas that the author would like to explore and develop in the very near future. First, modify the CART to work in a networked environment. Second, develop similar tools for other *NIX operating systems.

Implementing the "CART" in a Networked Environment

To implement the CART in a networked environment running NIS or NIS+ and NFS would have been much simpler. The "control host" could be placed on the network, and if a naming service is running, then all hosts will know about each other. Three types of CART servers can be placed on the network: (1) a standalone CART server; (2) a JumpStart server providing only CART functionality; and, (3) a JumpStart server providing both CART functionality and JumpStart functionality. A method for altering the BEGIN script to perform the restore (clone) can be implemented fairly easily. Thus, the JumpStart server can be used to perform site-specific JumpStarts as well as the CART method of imaging. Instead of maintaining the images of "target hosts" on optical media, the images can be stored on network disk space and accessed through NFS. Again, this would be an entirely different way to unleash the power of JumpStart.

Implementing the "CART" in Other *NIX Operating Systems

One of the obvious next steps is to investigate implementing this tool in other *NIX operating systems such as HP-UX, IRIX, AIX, Digital UNIX, FreeBSD, BSDI, and Linux. The first operating system to tackle on the list will be HP-UX, since it already has the fully developed Ignite-UX utility.

Limitations of the "CART"

1. The biggest limitation to the CART is encountered when the target host contains relatively large filesystems (greater than 1 GB) and CD-R

is chosen as the optical media to store the system image. The limitation lies in the maximum carrying capacity of a CD-R which is 650 MB. By using compression utilities (compress or gzip) in this tool, and assuming about 60% compression efficiency, the maximum filesystem size that can be captured is about 1000 MB or 1 GB. So, when capturing to CD the system image of a target host that contains at least one filesystem larger than 1 GB, the CART will report an error. The error is due to the resultant compressed file for the filesystem exceeding the carrying capacity of the CD. However, there still is hope for systems containing filesystems larger than 1 GB. The system image could and should be captured to DVD-R, since the carrying capacity of a DVD-R media is about 4 GB. Again, assuming about 60% compression efficiency, a filesystem on the order of about 7 GB can be imaged.

2. A system image saved to a multi-volume set requires an additional optical media drive for the restore operation to work.
3. The system being restored must not only have the same kernel architecture but also the same platform architecture as the system that was imaged. As an example, even though an Enterprise 250 and an Enterprise 450 have the same kernel architecture (sun4u), they have different hardware platforms. When the image from one is restored onto the other, the resultant system will not be bootable.
4. When restoring a system or cloning to a like system, the disk drives in the host being restored must be as big or bigger than the disk drives in the original host imaged.
5. Currently, the CART cannot capture system images of target hosts that have mirrors or RAID volumes (created by Veritas Volume Manager or Solstice DiskSuite) on disk drives larger than 1 GB. When such logical volumes exist, the entire disk must be captured using the dd utility to ensure the private region on the disk is captured. By having to capture the entire disk as one image file, even after compressing, the resultant file would not fit on one CD volume. This limitation goes away if the CART is implemented in a networked environment.
6. Currently, the CART is only implemented for the Solaris operating system.

Additional By-Products That Arose From the "CART" or Its Technology

Several by-products resulted from the development of this tool. The possibilities are wide open for many more. The more useful of these by-products are identified below and how they were accomplished.

Make Copies of the Solaris Install CD. How many times have you not been able to locate your Solaris Install media because there were too

few to begin with and they are all checked out (or locked in the office of the other system administrator)? Well, now you can make as many copies as you like (keeping in mind copyright laws). Make a slight modification to the Bourne shell script used during Phase 1 that copies select portions of the Solaris Install CD to the "image disk," to have the script copy the entire CD. (Actually, during the initial stages of developing the CART, the original script copied the entire Solaris Install CD to the "image disk" anyway.) Following the completion of the copy, go straight to the burn phase (Phase 3). While the "image disk" is in this state, as many copies that are desired can be burned off, one at a time. Using the CART configuration described earlier in this paper, it took about 30 minutes to copy the entire Solaris Install CD to the "image disk" and about five minutes to burn off a copy.

Make A Customized Solaris Install CD or CD Set. Instead of having the generic Install CD as provided by Sun Microsystems, place some of your Customized JumpStart configurations on the CD. This essentially makes your JumpStart come off of the CD instead of a JumpStart server. Who would want to do this? How about an experienced JumpStart administrator who finds him/herself in a non-networked environment and is faced with a task of rolling out a large number of host installs or upgrades. If there are several additional packages and tarballs needed for your intended JumpStart, additional CDs may be required. By using the same technology incorporated into the CART, having additional CD volumes will not be a problem. However, just like with the CART, an additional CD drive will be needed to perform the JumpStart from CDs.

Make A Specialized Bootable CD. An industry authority on UNIX Backup and Recovery, (W. Curtis Preston) has asked to have a bootable CD built that will contain enough of an operating system to allow third party backup and recovery software (Legato Networker and Veritas NetBackup) to run. Very customized scripts automatically invoked on the CD will prompt the user for certain information. Based on the information provided, the backup software selected will proceed to perform backups over the network to restore a complete system image. The intent of making the CD bootable is for the situation of a boot disk crash. For such situations, this specialized tool will perform a bare-metal recovery (disaster recovery on a virgin disk drive) and will result in bringing the system to the state of its last good backup. The development of this tool is currently in progress and looks quite promising.

Resources

The following freeware products from Joerg Schilling [9] were used in the development and implementation of the CART:

cdrecord	A program for creating single/multiple session CD-R on a SunOS, Solaris, Linux, *BSD/SGI, HP-UX, AIX, NeXT-Step, or Apple-Rhapsody system.
sformat	A program to format/analyze/repair SCSI hard disks on a SunOS, Solaris, or Linux system.
scg	A driver to send any SCSI command to any SCSI device on a SunOS or Solaris system.
fbk	A driver to mount a file containing a filesystem; (File simulates Block device on Solaris).
mkisofs	Puts files in ISO-9660 format.

A "Smart and Friendly" CD-RW 426 Deluxe CD-Recorder was used in the development and final implementation. No issue arose with the use of this CD-RW device during the development and use of the CART.

Other CD-R recording hardware and software products (i.e., Young Minds, Inc., HyCD, Gear to name a few) could have been integrated into the CART as well. However, the price of "cdrecord" and its associated products could not be beat. There were not any issues encountered with the installation or use of the "cdrecord" products or with the use of the "Smart and Friendly" CD-RW 426 Deluxe CD-Recorder. Both of these products receive a high endorsement from the author.

Acknowledgements

First, I thank the *Collective Intellect* of Collective Technologies who provided an invaluable resource and wealth of knowledge on many areas during the development of the CART.

Second, I thank Joerg Schilling. Indispensable in the creation and final product of the CART were several shareware products provided by Joerg Schilling.

Third, I thank Adelaida Esquivel, Senior Computer Programmer, who spent countless hours supporting the process of writing and provided the proofreading of this paper.

Author Information

Lee "Leonardo" Amatangelo was graduated from the University of California, Irvine in 1983 with a B.S. in Molecular Biology and in 1985 with a B.A. in Anthropology. He has been working in the computer industry since 1981. Currently, he is a systems management consultant specializing in Solaris and disaster recovery for Collective Technologies. He can be reached via email at leonardo@colltech.com and by

physical mail at Collective Technologies, 9433 Bee Caves Road, Building III, Austin, TX 78733.

References

- [1] Sun Microsystems, *Solaris 2.6 – Solaris Advanced Installation Guide*, Mountain View CA, Part No. 802-5740-10, August 1997, Revision A).
- [2] Heiss, J., “Enterprise Rollouts with JumpStart,” *LISA XIII Conference Proceedings*, 1999.
- [3] Kasper, P. A. & McClellan, A. I. *Automating Solaris Installations – A Custom JumpStart Guide*, SunSoft Prentice Hall, 1995.
- [4] Nemeth, E., Snyder, G., Seebass, S., & Hein, T., *UNIX System Administration Handbook*, 2nd Edition, Prentice Hall, 1995, Ch. 9.
- [5] Shaddock, M. E., Mitchell, M. C., & Harrison, H. E., “How to Upgrade 1500 Workstations on Saturday, and Still Have Time to Mow the Yard on Sunday,” *LISA IX Conference Proceedings*, 1995.
- [6] Zuberi, A., “JumpStart in a Nutshell,” *Inside Solaris*, February 1999, Ch. 1
- [7] <http://dvddemystified.com/dvdfaq.html>.
- [8] [http://www.fadden.com/cdrfaq/faq00.html#\[0-1\]](http://www.fadden.com/cdrfaq/faq00.html#[0-1]).
- [9] http://www.fokus.gmd.de/research/cc/glone/employees/joerg_schilling/private/.
- [10] <http://www.gearcd.com/>.
- [11] <http://www.hycd.com/>.
- [12] <http://www.pioneerusa.com/>.
- [13] <http://www.smartandfriendly.com/>.
- [14] <http://www.ymi.com/>.

A Linux Appliance Construction Set

Michael W. Shaffer – Agilent Laboratories

ABSTRACT

Open source UNIX-like operating systems offer unique opportunities for administrators to create appliance style operating system and application packages that meet their own specific needs. This paper provides examples of several commonly useful appliance configurations based on the Linux operating system and documents the motivations, principles, and techniques behind the development of a basic 'Linux Appliance Construction Set' known as LxA which was used to produce them. Experience with LxA so far suggests that its use may help system administrators significantly reduce the amount of time spent deploying, maintaining, upgrading, and documenting certain types of hosts under their control.

Introduction

This paper details the philosophy, tools, techniques, and sample implementations that I have created in the course of developing a Linux Appliance Construction Set which I refer to as LxA. Certainly the philosophy and many of the the general principles detailed herein would apply to the construction of similar systems based on any open source UNIX-like operating system, but I chose Linux because of my long personal experience and familiarity with it. LxA expands to: Linux x Appliance, for instance: LRA abbreviates Linux Routing Appliance.

The original design parameters for LxA included the requirement that the entire boot and root file system images fit on a single 3.5" floppy disk. The reader might legitimately wonder why I would embark on the creation of yet another mini Linux when there are already a number of excellent and similar projects active in the open source community [17, 11, 4, 10]. While most mini-Linux distributions aim to provide completely functional systems on low resource platforms, LxA attempts only to provide a single or small number of well defined features per configuration. It is not so much the minimal size as the minimalist philosophy that distinguishes LxA from its peers. In fact, some configurations such as LPA-CD (Linux Printing Appliance on CD-ROM) are quite large and require a CD-ROM for their root file system. Since the initial release of LxA, other community projects have arisen which pursue principles similar to those of LxA with apparently excellent results so far [5].

In general, LxA seems to be well suited to systems such as: Internet connection sharing routers, firewalls, bridges, routers, print servers, certain types of file servers (such as CD-ROM towers), terminal servers, point of sale terminals, browser kiosks, or in general any type of small to medium size single purpose system. LxA would probably not serve well for systems such as desktop 'power user' or developer workstations whose users regularly install, test, and uninstall many applications. LxA is also not intended to serve as a general purpose or standalone Linux

distribution although it is hoped that the sample configurations will serve well as a basis for the future development of many different types of Linux appliances.

Motivation

As a system administrator, I frequently work part-time for small businesses, organizations, and departments who don't for one reason or another have in-house system administration staff. When, in mid 1999 I decided to move from South Carolina to Silicon Valley, I was faced with the dilemma of supporting a number of existing installations of Linux machines providing file, print, and network routing services which I would now be able to visit in-person perhaps only once a year. While Linux has inherited both a well deserved reputation for reliability and excellent facilities for remote administration from its UNIX ancestors, there are still situations which are extremely difficult to manage remotely. Furthermore, even with the recent surge in popularity that Linux has enjoyed it can still be difficult to find local companies or individuals who can provide reliable and affordable UNIX system administration to small customers in many regions.

A typical serious problem that I considered facing was the catastrophic failure of a hard disk or power supply in a critical machine such as a print server. While it would be easy to remotely coordinate procurement and installation of replacement hardware for such a failure, the rebuild or restoration of the software load for even a simple Linux server could be quite a dicey undertaking if trusted to inexperienced personnel working three-thousand miles away. I considered the possibility of having replacement equipment shipped to my new location, configuring it myself, and then shipping it on to its destination, but I felt that this approach was undesirable since it would at least double the already significant costs, delays, and risks inherent in shipping computer sized pieces of electronic equipment around the mainland U.S. In addition, there was always the risk that I might make

some trivial error in configuration that would only be discovered when the system was powered up in its final destination and failed to work properly (worst of all would be a mistake that also prevented me from remotely accessing the machine).

In response to these considerations I formulated the idea of building Linux systems which would require no installation in the traditional sense but which would simply boot and run directly from the media on which they were delivered. My initial goal was to create fully functional internet connection sharing and print spooling systems contained entirely on a bootable floppy or CD-ROM. As I progressed in this endeavor, I also realized that I was unhappy with the significant amount of time that was required to 'adjust' the typical Linux distribution to my own purposes and tastes after installation, and I decided to enlarge the scope of the LxA project to include development and documentation of a general purpose framework for building small Linux systems tailored precisely to the needs of my typical customers. Such Linux systems would allow me to provide customers with backup copies of the software load for their systems by simply providing extra copies of their media upon delivery and would also allow me to deploy upgrades and fixes to systems much more economically by simply shipping new CDs or floppies to all affected customers. Thus it was that I embarked upon the creation of LxA with the following goals:

- Reduce system setup time
- Reduce system maintenance and upgrade time
- Reduce system complexity
- Reduce probability and impact of hardware failure

These are elaborated below.

Reduce System Setup Time

The typical procedure for setting up a new host using a general purpose operating system involves performing an installation from original media and then applying a long series of patches, packages, and procedures to adjust the system to the particular level of functionality and security desired. Certain operating system projects such as OpenBSD [14] seek specifically to minimize the number of unnecessary components in a default installation and to dramatically reduce the number of steps required to achieve a high level of security after initial setup. Other projects such as Bastille Linux [1] do an excellent job of distilling and codifying community knowledge of the best practices for a given platform into hardening scripts which aid administrators both in learning and applying a consistent security policy to new systems. LxA, on the other hand, attempts to provide a set of procedures and examples for quickly composing new systems based on minimal filesystem trees that include exactly the components required in the proper configuration. Once a prototype system tree is created and tested, it can be deployed to new systems much more rapidly and with greater confidence in its correctness and security.

Reduce System Maintenance And Upgrade Time

Many individuals and organizations seem to forget that most of the time spent on any given computing system will be spent maintaining it, not designing or installing it. For this reason, LxA seeks to reduce to an absolute minimum the time and problems involved in upgrading systems with new software. The typical LxA system will have its root filesystem and configuration files on one or more pieces of removable media, so upgrading the system can be accomplished instantly and reliably by simply halting the system and replacing this easily handled media.

The boot and configuration filesystems can be extensively tested before distribution to remote locations to ensure a smooth upgrade, and in the event of a malfunction it is trivial to just replace the old media and restore the system to a working state until further troubleshooting can be performed. The desire to facilitate this rapid and easy roll-forward and roll-back of system configurations was a primary incentive to the development of LxA.

By avoiding completely the typical hard-disk based installation procedure, LxA systems also encourage rapid deployment of updated software once it has been tested, since mass upgrades and downgrades involve drastically reduced time and risk of service interruption. Additionally, the backup of the system binaries and configuration files of an LxA system becomes almost trivial since it means simply producing and storing one or more copies of the boot and root disk images before distribution. This type of backup allows almost instantaneous recovery to known good states both from system failure and from system compromises or intrusions. Disaster recovery becomes simply a matter of obtaining suitable hardware instead of the time consuming and error-prone process of rebuilding a complete system from whatever notes are available (or even worse, from memory).

Reduce System Complexity

This was one of the primary motivations behind the early development of LxA systems. One of my original targets was to produce a working and useful Linux system composed of the absolute minimum number of pieces and, in the process, to gain an understanding of exactly what each piece provided and required. A second motivation was to reduce dramatically the amount of time and effort needed to thoroughly document the configuration and function of critical systems. All too often documentation is neglected or completely forgotten due to the time and effort required both to research and prepare it.

One can easily see that a Linux based firewall system such as LFA (Linux Firewall Appliance) composed of only around 40 files is far easier both to understand and to document than one based on a typical general purpose distribution which might include hundreds or even thousands of files after even a

minimal installation. It is my belief that administrators are most likely to produce reliable and secure systems when they thoroughly understand all the components involved, and LxA seeks to facilitate this understanding by dramatically reducing system complexity.

Reduce Probability And Impact Of Hardware Failure

A typical small LxA system such as LFA will often boot from a floppy and will run entirely from a RAM disk once started. In such minimal configurations, there is no need for either a hard drive or a CD-ROM. This obviously eliminates at least two components which can fail, and the omission of these devices will also aid the longevity of the system power supply by reducing the strain placed on it. For systems such as LPA-CD with larger filesystem space requirements, it is still possible for the system to operate without employing a hard drive, depending on the application.

In some cases a fixed disk is the most efficient solution to certain requirements such as the need for swap or spooling partitions, but even in these situations the system can still be upgraded without the need to manage the internal storage. LxA systems typically feature some simple procedures in their startup scripts to optionally format, mount, and populate any necessary swap or spooling areas at boot time, and LPA-CD includes examples for using RAM disks, network filesystems, or fixed disk partitions for these purposes. If desired, LxA systems can even partition their own fixed disks at boot time through the use of a non-interactive partition editor such as `sfdisk` in their startup scripts. These features facilitate easy and rapid replacement of failed parts or even whole machines since LxA configurations can be easily tuned for no-install or self-install behavior on new hardware.

Principles

In pursuit of the goals stated above, LxA systems are generally designed to adhere to the following principles:

- Build systems by composition, not reduction
- Run from read-only and/or removable media
- Omit login and run-time configuration
- Use modern and standard software components

The next sections detail these principles.

Build Systems by Composition, Not Reduction

This means developing a set of practices for identifying the minimal set of components required for any given capability and then adding exactly those components to a base or minimal LxA prototype image.

Run From Read-only and/or Removable Media

Although some LxA configurations make extensive use of fixed disk partitions for `/var`, `/tmp`, and swap areas, the general rule is to keep all system binaries and configuration files on removable and preferably read-only media. The boot and root partitions may be made effectively read-only by loading them

into RAM disks at run time, mounting devices such as floppy disks in read-only mode, or using physically read-only media such as write-once recordable CDs. While floppy and CD-ROM media are readily accessible and familiar to many users, some applications may benefit from the employment of alternative boot and root devices such as flash ram drives or cards.¹

Omit Login and Run-time Configuration

This principle not only helps to save valuable kilobytes on boot media, but also makes the system much more tamper-proof in potentially hostile environments. Obviously this approach also requires the administrator to perfect the system configuration during staging. This concept perhaps more than any other distinguishes LxA from other small Linux systems such as the Linux Router Project [11]. Certainly this approach will not work well for all applications, but a machine configured in this fashion is a true network appliance, running only exactly what it needs to perform its designated functions. While console login facilities are often omitted in production LxA configurations, the example packages also include simple procedures for installing a minimal set of interactive tools and a console shell on the system for troubleshooting during staging.

Use Modern and Standard Software Components

In contrast to other mini Linux distributions, LxA uses the most modern and standard components available wherever possible. The Linux Router Project (See [11]), for example, employs the Linux 2.0 kernel, `libc5`, and the BusyBox POSIX tools package [2] to provide a wide range of capabilities while maintaining the smallest possible disk and memory footprint. Certainly there are a number of good reasons for using older components including smaller binary size, greater maturity, and often a more sedate pace of development. In fact, one of the greatest advantages of

¹Discussions with LxA users during the writing of this paper led to the idea of building LPA systems based on a combination of a 48MB CompactFlash card, a CompactFlash to IDE adapter available from the Tuscon Amateur Packet Radio organization (See [16]), and commonly available hot-swap hard drive carriages for standard size hard drives. This configuration should feature higher reliability than either CD-ROM or floppy based systems while still affording ample capacity for typical configurations. As of this writing, 8-128 MB CompactFlash cards are available at reasonable price points. This option requires little modification to existing LxA configurations since the TAPR device makes a CompactFlash unit appear to a PC system as a standard IDE drive, and the benefits of easy system upgrades will be preserved since the removable hard drive carriage will make it easy to swap flash cards. Finally, this style of system would have the benefit of somewhat higher physical security than CD or floppy based configurations since many swappable drive carriages feature key locks, and the logistics of media distribution may even benefit slightly from the much smaller size and greater ruggedness of CompactFlash compared to floppy disks. Complete details of the outcome of this effort will be documented on the LxA project site [12].

using open source tools in my experience is that stable versions of products tend to remain viable for a very long time due precisely to the availability of source code that can be compiled for older platforms even when binaries are no longer readily available.

Nevertheless, I decided to avoid the use of older and non-standard components wherever I could. When pressed for space on small capacity boot media such as floppy disks, my approach was to save kilobytes by eliminating components entirely rather than recompiling, patching, or down-grading them. Usage of an older kernel and C library requires that all components be compiled specifically for that environment, and I wished to avoid forcing users to compile anything just to get a new LxA system running. Also, while older or customized components would undoubtedly save significant amounts of space on LxA systems, the benefits of using them would become completely negated if I wished to add a tool or service that depended on more modern and bulkier components such as glibc.

This principle represents a significant difference in philosophy between LxA and several of the other mini distributions available. LxA provides not so much a specific distribution of Linux as a framework for assembling this sort of mini system from any current Linux distribution at hand. Using standard components allows more users to modify the system in unforeseen ways and should also allow LxA to track new developments in the Linux kernel, major libraries, and applications much more closely than distributions which rely on heavily patched or custom components.²

Issues

Remote Administration

Since LxA configurations consistently use standard binaries and libraries from more conventional Linux distributions, they frequently leave little space on small media such as floppy disks for remote or even local administration tools. The floppy based configurations such as LPA (Linux Printing Appliance) simply have no space for components such as `sshd` or `inetd`. In fact, in the case of LPA, there is no room for even a minimal shell environment. In comparisons with other Linux distributions, even those of the mini variety, this could be considered a serious disadvantage for LxA.

²Even as I was finishing this paper, I was working on upgrading all the LxA example configurations to linux kernel version 2.2.17, glibc 2.1.3, and the latest versions of all the binaries included. The total effort involved was a couple of kernel compiles, some work with `find` and `cp` to copy new binaries into the root filesystem trees, and re-making of the distribution tarballs. Other projects such as the Gibraltar firewall [5] have almost completely automated this process by employing scripts to find and assemble all required binary components on the fly from the machine on which the configuration is performed. Such an option would probably be a valuable addition to LxA and may be added in the future.

On the one hand, I consciously decided up front to eliminate console and remote login capabilities wherever possible both for increased security and to free up disk space desperately needed for other things. On the other hand, for configurations such as LPA-CD, I have in fact included both a proper login shell and `sshd`³ to allow the possibility for remote administration if desired. So, the lack of remote administration facilities can perhaps be seen as either an advantage in disguise or as a non-issue depending on the configuration and applications involved. In some respects addressing this issue may simply require an adjustment of perspective. LxA was intended to address precisely those situations where remote administration capabilities are of little use, so what appears to be an advantage for other systems may in fact not be as significant as it seems in the environments for which LxA was designed.

System Logs and Spool Areas

The options LxA presents regarding boot-time management of fixed disks or the use of RAM disks for writable filesystems obviously raise the question of how to store persistent accounting information such as system logs. The typical LxA system is not one intended for hosting interactive logins, so it is assumed that most information of value would be included in the messages recorded by the system and kernel logging daemons. There is nothing to prevent an LxA system from using startup scripts that re-format the system log partitions only when absolutely necessary, and this would allow persistence of system logs for later analysis and troubleshooting in the event of problems. Alternatively, in certain environments it may be desirable to have LxA based machines send their log messages to one or more remote syslog hosts. Remote logging is often an excellent method for facilitating diagnosis of severe system failures or compromises and is commonly recommended in any event.

Another issue which may arise is that use of the automatic partition and format features would result in the loss of any pending jobs on filesystems such as print spool areas. In general, LxA simply leaves it to the individual administrator to evaluate their application for a system such as LPA-CD and determine what balance of resiliency, persistence, and hardware requirements fits their environment best. For example, the following three scenarios might be considered when deploying print spooling and format conversion servers based on LxA:

1. Booting from a CompactFlash device and using only RAM disks for spooling areas. This option would require careful consideration of the anticipated printing load, especially if a significant number of PostScript jobs would be spooled to non-PostScript printers. A configuration for moderate to heavy loads might benefit from the inclusion of 512-1024MB of RAM.

³`Sshd` will be a standard component of LPA-CD by the time this paper is published.

This would allow an ample 128-256MB for working set and leave 384-768MB for a /var filesystem on a RAM disk.

While this might at first seem like an extravagant amount of memory, consider that the result would be an entirely solid state print server that would likely perform extremely well, limited only by the speed of its CPU. Recent trends in hardware prices would make this only a moderately expensive configuration for most organizations. The obvious disadvantage of this configuration is that it makes no allowance for persistent spool areas, meaning all pending jobs would be lost in the event of a power failure or other serious event. In some cases this disadvantage may well be outweighed by the ruggedness, reliability, and extremely low maintenance requirements that such a configuration would present. This would probably work well for distributed print servers that need to be scattered throughout a building, campus, or region with little on-site system administration attention.

2. Booting from CD-ROM and using local hard drives for spooling, log, and swap area. This option is probably the easiest for the typical UNIX system administrator to evaluate and provision, and would probably work well in the a computer room or departmental environment where the desire is simply to have a standard and easily replicated configuration. Despite its conventional approach, this configuration still presents some options with regard to manipulation of the fixed disks at start up time. Simply mounting a pre-formatted and populated /var filesystem provides for a simple startup and complete persistence but requires that spool disks be partitioned, formatted, and populated manually at setup time or in a separate machine. Automatic partitioning and formatting, on the other hand, would provide for extremely quick setup and allow instant replacement of failed spool disks, and this might be a good choice if the cost of loosing spooled jobs and log files is felt to be outweighed by the consequent reduction in maintenance effort.
3. Booting from CD-ROM and using NFS exports for spooling and log areas (with perhaps file based swapping over NFS as well). This option features a mix of advantages and disadvantages from each of the two options above. On the one hand, it relies on having a network infrastructure already in place including at least one machine providing exported filesystems over NFS. On the other hand, if such an infrastructure is available, this option allows the simple startup and complete persistence of the second option, permits a completely solid-state hardware configuration for the print servers

themselves as in the first option, and doesn't require either a huge amount of RAM or any local hard disks to be procured for individual print servers. The critical element to consider in evaluating this option is obviously the reliability and speed of the network and NFS file services available. Across a computer room, building, or even a small campus environment with widespread switched 100Mbit ethernet networks, this option could work nicely even for distributed print servers. An obvious disadvantage of this configuration is that a failure of the spool area file server or its network segment could potentially cripple a large number of print services, but the same could also be said of the centralized print spooling servers used in many more conventional arrangements.⁴

The main point the reader should note from these examples is that LxA by design makes choosing or even switching between all three of these alternatives almost trivial. Leaving aside the issue of hardware re-configuration for a moment, consider the effort that would be required to configure a typical Linux distribution for either the first or third options, and then consider that with LxA choosing between them means changing just a few lines in one startup script. The whole intent of LxA is not to dictate or assume a certain pattern of operation but to make many different options available with as little effort as possible.

Older Hardware

While the low resource requirements of LxA systems may allow the use of hardware that would otherwise be considered obsolete, it is important to note that older hardware is often more prone to failure and, more significant for many organizations, no longer supported by warranties or service agreements. On the other hand, the upgrade cycle for end-user systems in many organizations makes available a significant number of systems each year that are due for retirement. These systems increasingly often pose an expensive and troublesome disposal problem for their owners. Given the right application and software load, some of these machines can be successfully re-purposed as departmental, standby, or load-sharing

⁴Upon reflection, the reader may reasonably ask what advantage there could be in this option over a more conventional, centralized print server. After all, if anything, this option introduces *more* hardware rather than less into the picture when compared to using a single massive machine for the job. One example might be a situation where an organization wishes to make a number of existing non-network and/or non-PostScript capable printers available for network printing throughout a building or campus. If a sufficient number of existing obsolete machines with network cards and CD-ROM drives are available, a configuration like this would allow upgrading these printers with both network direct and PostScript capabilities by connecting them via serial or parallel ports to LPA-CD machines distributed to wherever the printers physically reside.

servers complete with a ready supply of spare parts from any unused machines in the pool to counter the greater potential for failure and lack of support.

Tools and Techniques

I decided when creating LxA to start with an empty system and add features through composition instead of trying to cut down an existing distribution or system through a process of reduction. This approach, I felt, would result in a smaller, easier to understand system with less development time. I wanted to create and document a system in which I could explain the purpose of every single file and process. In addition to learning more about Linux and UNIX systems through this exercise, I hoped to also develop a toolbox of general principles and techniques for building other minimal systems in the future.

Assembling Services

This is a general description of the tools and techniques I use to figure out the necessary components for a given service. I usually begin, of course by locating and examining the binaries for the service I want. For this example I've chosen to use Samba, the NetBIOS file and print service for UNIX-like systems. I always start by installing and configuring the desired service on a staging machine before I try to deconstruct it for a mini system. Of course, any available man pages, web sites, mailing lists, and other available documentation for the service in question should be considered required reading throughout this process.

Distribution Package Managers

Most Linux distributions employ some sort of package management tools for installing and removing software. The two most commonly used package management tools are rpm (the RedHat Package Manager) and dpkg (the Debian PacKaGe manager). Distributions employing rpm include RedHat, SuSE, Mandrake, and many others. Debian, Corel Linux, Stormix, and a few others use dpkg.

The maintainers of package files have usually put a great deal of work into determining the dependencies of each particular package, so I will start there if a distribution package is available for a particular service. The first step is to use the package manager's 'list files' function to see exactly what files are

installed on the system by a particular package. To query for the files installed by the rpm package for Samba, I would employ a command such as:

```
# rpm -ql samba
/etc/logrotate.d/samba
/etc/pam.d/samba
/etc/rc.d/init.d/smb
/etc/smbusers

...
/usr/doc/samba
...
/usr/sbin/nmbd
/usr/sbin/samba
/usr/sbin/smbd
/usr/sbin/swat
/usr/share/swat
...
/var/lock/samba
/var/log/samba
/var/spool/samba
```

Newer dpkg based distributions will often split services and their clients up into several packages to minimize dependencies and resource requirements for each package, so I start by querying the package management database to see what installed packages might be related to Samba; see Listing 1. This reveals four packages, and I can tell from the descriptions that I am probably only interested in the first two for the sake of an LxA system. To query for the files installed by each Samba package, I would next invoke dpkg with the -l option for each interesting package:

```
# dpkg -L samba
/.
...
/usr/sbin/smbd
/usr/sbin/nmbd
...
/var/samba
...
/etc/init.d/samba
/etc/cron.daily/samba
/etc/cron.weekly/samba
# dpkg -L samba-common
/.
...
/etc/samba/codepages
...
```

```
# dpkg -l '*samba*'
```

```
Desired=Unknown/Install/Remove/Purge/Hold
| Status=Not/Installed/Config-files/Unpacked/Failed-config/Half-installed
|/ Err?=(none)/Hold/Reinst-required/X=both-problems (Status,Err: uppercase=bad)
||/ Name Version Description
+++-----+-----+-----+
ii samba 2.0.7-3 A LanManager like file and print server for UNIX
ii samba-common 2.0.7-3 Samba common files used by both clients and server
ii samba-doc 2.0.7-3 Samba documentation.
pn task-samba <none> (no description available)
```

Listing 1: Package management query: samba.

```

/etc/samba/smb.conf
...
/etc/pam.d/samba
...

```

In general, for the purposes of an LxA system, I ignore any files installed under `/usr/doc`, `/usr/share/doc`, `/usr/man`, `/usr/share/man`, and `/usr/local/man`. These files are simply the documentation and man pages and are not necessary for functioning of most services. Other files which are probably not necessary are headers placed in `/usr/include` or `/usr/local/include`. Files I give the most attention to are those placed in `/etc`, `/bin`, `/sbin`, `/lib`, `/usr/bin`, `/usr/sbin`, `/usr/lib`, `/libexec`, `/usr/libexec`, and `/usr/share` (outside of `/usr/share/doc` or `/usr/share/man`). Many services will install cron jobs (typically in `/etc/cron.daily` or `/etc/cron.weekly`), and these may or may not be necessary for an LxA system. Many times these cron jobs serve only to rotate logfiles, and such actions are not necessary for example when using a remote syslog host. On RedHat the files installed under `/etc/logrotate.d` may also be safely ignored for similar reasons unless the `logrotate` utility will be used on the mini system. In the example shown here, the RedHat Samba package includes the `swat` web based administration tool for Samba, whereas the Debian package seems to leave it to a separate installation. The decision to include or exclude utilities such as these will depend mostly on the size constraints of the boot media for the target system and the anticipated usefulness of the individual tools. Since I hardly ever use `swat` for managing Samba machines and it presents significant space requirements, I chose to omit it.

The `rpm` and `dpkg` package managers, in addition to providing information about specific packages, will usually also provide a list of other packages which a service requires or suggests for proper operation. To query for the description and dependencies of a package using `dpkg`:

```

# dpkg -p samba
...
Depends: samba-common (= 2.0.7-3),
        libc6 (>= 2.1.2), libncurses5,
...

```

Once again, if the specified dependencies are not already present on the LxA base system image, they must be located and added. The Debian package manager lists dependencies in terms of other packages that must be installed, whereas the RedHat package manager may list either other packages or individual files. Both package managers provide version information for dependencies as well; stating whether each required package must be of some exact version or simply greater than a certain version for compatibility. If a dependency is itself a complex package, it may be necessary to repeat the process of exploration until all levels of dependencies have been satisfied. An example command for listing dependencies using `rpm` would be:

```

# rpm -q --requires samba
pam >= 0.64
samba-common = 2.0.6
/sbin/chkconfig
/bin/mktemp
/usr/bin/killall
fileutils
sed
/bin/sh
ld-linux.so.2
libc.so.6
libcrypt.so.1
libdl.so.2
libnsl.so.1
libpam.so.0
libreadline.so.3
libtermcap.so.2
/bin/csh
/bin/sh
/usr/bin/awk
libc.so.6(GLIBC_2.0)
libc.so.6(GLIBC_2.1)

```

Both of the package manager listings included a dependency on a package named `samba-common`, so further exploration of this package would be necessary in this case.

Although the package manager file listing reveals a large number of accessories and configuration tools, I know from experience with this service that the actual file and print service portion of Samba needs only two basic binaries to function, namely `smbd` and `nmbd`. Unfortunately, I have found that in making this sort determination, there is little substitute for experience and careful observation, although well documented services may prove easier to analyze than those which are not. As a final step before copying the binaries into my LxA image tree, I use the `ls` command just to check if they have any unusual permissions or modes set which may need to be preserved:

```

# ls -al /usr/sbin/*mbd
-rwxr-xr-x 1 root root 338092
              Jul 27 03:56 /usr/sbin/nmbd
-rwxr-xr-x 1 root root 738556
              Jul 27 03:56 /usr/sbin/smbd

```

In this case there appears to be nothing out of the ordinary.

Ldd

After an initial survey using the package manager, man pages, and `ls`, I typically use `ldd` to see what system libraries the service components are linked with and where they are located:

```

# ldd /usr/sbin/*mbd
/usr/sbin/nmbd:
  libdl.so.2 => /lib/libdl.so.2
              (0x2aac3000)
  libcrypt.so.1 => /lib/libcrypt.so.1
              (0x2aac7000)
  libc.so.6 => /lib/libc.so.6
              (0x2aaf4000)
  /lib/ld-linux.so.2 => /lib/ld-linux.so.2
              (0x2aabb000)

```

```

/usr/sbin/smbd:
libdl.so.2 => /lib/libdl.so.2
              (0x2aac3000)
libcrypt.so.1 => /lib/libcrypt.so.1
              (0x2aac7000)
libc.so.6 => /lib/libc.so.6
           (0x2aaf4000)
/lib/ld-linux.so.2 => /lib/ld-linux.so.2
                   (0x2aaab000)

```

If the libraries listed are not already present in the /lib directory of the LxA configuration I am working on, I would use something like:

```

# cd /lib
# ls -al libcrypt*

-rw-r--r-- 1 root root 20340 Nov 1
13:56 libcrypt-2.1.2.so
lrwxrwxrwx 1 root root    17 Nov 9
19:03 libcrypt.so.1 -> libcrypt-2.1.2.so

# find libcrypt* | cpio -pvd \
    /usr/local/linutopia/root/lib

```

to add them. I recommend using `find` and `cpio` instead of `cp` since these tools usually do better job of preserving permissions, modes, and linkages.

Lsof

If problems arise or I suspect that the service requires more than just the explicitly linked libraries shown by `ldd`, I will try executing `lsof` while the selected service is running to view a comprehensive list of file handles, sockets, and other resources the service has open. This command usually produces rather lengthy output, so I have snipped some irrelevant details from the example in Listing 2. I observe

from this listing that both daemons are accessing another library which `ldd` didn't say anything about, namely `libnss_files`. They also appear to have some pid files and such open in `/var/samba` and `/var/log`, so I would make a note to make these directories available in the `/var` filesystem of the mini system. Finally, they have some network ports open that they are listening on, namely `netbios-ns`, `netbios-dgm` and `netbios-ssn`. I would certainly take these into account when working out any `ipchains` rule sets for the mini system.

Once the binaries, all required libraries, and any working directories are in place in the filesystem tree for the mini system, I would copy the `/etc/samba` directory contents over to the new `/etc` directory and modify them appropriately. When I really get stuck trying to figure out what obscure file or directory a daemon is missing in the mini configuration, I usually fall back on the old faithful `strace` utility. While this tool produces even more profuse and cryptic output than `lsof`, it is often the only means of identifying exactly what resources a process uses. Many utilities will open files or devices only briefly during startup or other phases of their operation, and this can sometimes foil efforts which rely solely on the snapshot view of system handles provided by `lsof`. The output from `strace` is a complete 'diary' of all system calls made by a process, and usually looking for calls to `open()`, `stat()`, `read()`, `write()`, `socket()`, `connect()`, `send()`, `recv()`, `mmap()`, and `munmap()` will reveal all the filesystem and network objects in use by the traced process. A typical invocation of `strace` would be:

```
# strace -o smbd.txt /usr/sbin/smbd -D
```

```

# lsof | grep mbd

nmbd 405 root txt REG 3,66 338092 41150 /usr/sbin/nmbd
nmbd 405 root mem REG 3,66 85238 4116 /lib/ld-2.1.2.so
nmbd 405 root mem REG 3,66 10224 4167 /lib/libdl-2.1.2.so
nmbd 405 root mem REG 3,66 20340 4163 /lib/libcrypt-2.1.2.so
nmbd 405 root mem REG 3,66 936696 4119 /lib/libc-2.1.2.so
nmbd 405 root mem REG 3,66 32104 4178 /lib/libnss_files-2.1.2.so
nmbd 405 root 3w REG 3,66 804 923745 /var/log/nmb
nmbd 405 root 4ww REG 3,66 20 878636 /var/samba/nmbd.pid
nmbd 405 root 5u inet 3016 UDP *:netbios-ns
nmbd 405 root 6u inet 3018 UDP *:netbios-dgm
nmbd 405 root 7u inet 3022 UDP lazarus.hammes-chasn.com:netbios-ns
nmbd 405 root 8u inet 3024 UDP lazarus.hammes-chasn.com:netbios-dgm

***

smbd 407 root txt REG 3,66 738556 41149 /usr/sbin/smbd
smbd 407 root mem REG 3,66 85238 4116 /lib/ld-2.1.2.so
smbd 407 root mem REG 3,66 10224 4167 /lib/libdl-2.1.2.so
smbd 407 root mem REG 3,66 20340 4163 /lib/libcrypt-2.1.2.so
smbd 407 root mem REG 3,66 936696 4119 /lib/libc-2.1.2.so
smbd 407 root mem REG 3,66 32104 4178 /lib/libnss_files-2.1.2.so
smbd 407 root 3w REG 3,66 500 923748 /var/log/smb
smbd 407 root 4ww REG 3,66 20 878637 /var/samba/smbd.pid
smbd 407 root 5u inet 3030 TCP *:netbios-ssn (LISTEN)

```

Listing 2: Using `lsof` to find resources used.

This example would write its output to the `smbd.txt` file.

Source Code

Of course, one should never forget the one unique resource available to all users of open source software, namely the source code. While reading the source code for system utilities undoubtedly requires the most significant investment in time and effort of any technique discussed here, it also ultimately yields the most significant and longest lasting results in terms of experience, understanding, and confidence when practiced regularly. Based on my own experience, I steadfastly maintain that acquiring and practicing even basic programming and code reading skills will make any system administrator much more capable, flexible, and confident when dealing with new systems, unfamiliar problems, and unusual configurations.

Configuring a Read-only Root File System

Actually, a workable read-only root file system for most Linux applications differs little from the typical hard disk based root file system. LPA-CD uses a slightly different file system layout from many Linux systems, but that's only for the sake of simplicity. The specific things that I found which needed attention included:

The /var file system

Many processes place run time files in `/var`. Even a completely CD based system that uses no configuration floppy will still require at least a small RAM disk mounted on `/var` so that processes like `smbd` have somewhere to save their PID and temp files. Usually I will make `/tmp` a symlink to `/var/tmp` to keep all writable areas under one mount point. When an LxA system is configured to format and mount a `/var` file system automatically at each startup, the necessary files and directories may be populated within it either by using `tar` to unpack a skeleton image or by running a simple script of `mkdir`, `touch`, `chown`, and `chmod` commands. LPA uses the latter approach since it has no room for the `tar` binary whereas LPA-CD uses the former to make modifications easier.

Sockets in /dev

A few processes, notably `syslogd` and `lpd`, create sockets in the `/dev` directory to listen for their clients. Fortunately, `syslogd` poses little problem since it accepts a `-p` option specifying an alternative path for the socket that it usually opens at `/dev/log`. For LPA-CD, I typically start `syslogd` with a command line such as:

```
# syslogd -p /var/run/syslog.socket
```

I then create a dangling symlink from `/dev/log` to `/var/run/syslog.socket` in the read-only root file system to direct `syslog` clients to the real socket.

BSD `lpd`, on the other hand, presents considerably more trouble since it offers no command line argument to specify its socket path like `syslogd`. I was

forced in this case to break down and recompile the entire `lpr` package from source after changing a hard-coded `#define` in the `lpr` source code from `/dev/printer` to `/var/run/lpd.socket`. Again, I created a symlink from `/dev/printer` to `/var/run/lpd.socket` in case any clients outside the `lpr` package expected this socket to exist. When tracking down issues like this, I usually find `lsdf` and `strace` to be indispensable.

Various Files in /etc

The only files in `/etc` I know of that a Linux system tries to write to under normal conditions are `/etc/mstab` and `/etc/ioctl.save`. I took the easy way out with `mtab` and just made it a symlink to `/proc/mounts` which the kernel maintains automatically. This change allows `df` and `mount` to work correctly, and the system will simply discard anything written to `/proc/mounts`. The `ioctl.save` file is read and written by `init` and is used to set console parameters when the system enters single user mode. While `init` normally creates this file at boot time, it does not actually require it and will function perfectly well in its absence. All the other files typically found in `/etc` can either be read only or can just be symlinks to something on another file system if preferred. The one command that I haven't found any way to use with a read-only `/etc` directory is `passwd`. Because of the way `passwd` shuffles files around while changing an entry in `/etc/passwd`, it doesn't seem possible to use it successfully unless it can write to all of the following: `/etc`, `/etc/pwd.lock`, `/etc/npasswd`, and `/etc/passwd`.

If the `/etc` directory will be populated with symlinks, then there are exactly two files which must be physically present in this directory in order for `init` to start: `/etc/inittab` and `/etc/rc.init`. The `/etc/inittab` file cannot be relocated unless `init` is recompiled. Additionally, `/etc/inittab` must specify the path to the system startup scripts, and these scripts must exist on the root file system since nothing else will be mounted at this point. In the case of LPA-CD, I created an `/etc/rc.init` script which mounts the media for the `/local` file system so that all the dangling symlinks in `/etc` become valid. After running this script, LPA-CD's `inittab` hands off control to `/etc/rc.local`, which is actually a symlink to `/local/rc.local`, and this script handles the remainder of the system initialization.

Boot Process and Initrd Image

The Linux kernel contains support for a unique feature known as `initrd` or the Initial RAM Disk [9]. This capability is of critical importance to space constrained LxA configurations such as LPA since it allows a system to load and mount a small root filesystem image from a device that the kernel may not even contain drivers for. This mechanism requires only that the kernel have support for both RAM disk devices and `initrd` devices built in and places the burden on the boot loader to get the root filesystem image loaded and unpacked in memory along with the kernel. Thanks to this feature, the standard kernel for

floppy based LxA configurations is able to defer support for both floppy and ide devices to loadable kernel modules for a moderate savings in kernel size. The typical boot sequence for Linux systems using the initrd mechanism is:

1. BIOS loads first stage boot loader from a master boot record into memory
2. First stage loads and executes second stage boot loader
3. Second stage boot loader (usually lilo or syslinux) loads the kernel and initrd image into memory and uncompresses both
4. Linux kernel executes the script /linuxrc on the initrd image if present
5. Linux kernel creates the first userland process by executing /sbin/init
6. The init process executes system startup scripts (rc.init and rc.local for LxA)

The original purpose for the initrd mechanism was to allow booting from devices such as SCSI disks using a standard modular kernel without support for these devices built in. The idea was that the /linuxrc script could be exploited to load required driver modules from the initrd root filesystem image and then mount the real root filesystem from another device. LxA actually doesn't use the /linuxrc script at all but instead just runs the whole system from the initrd image. For the floppy based LxA configurations, the standard script for producing an initrd image from a root filesystem tree is:

```
#!/bin/sh
rm -f ./disk/initrd
rm -f ./disk/initrd.gz
dd if=/dev/zero of=./disk/initrd \
    bs=1024 count=$1
losetup /dev/loop0 ./disk/initrd
mkfs.minix -i $2 /dev/loop0
mount /dev/loop0 /mnt
cd ./root
find . | cpio -pud /mnt &> /dev/null
cd ../
umount /mnt
losetup -d /dev/loop0
gzip -9 ./disk/initrd
exit 0
```

Creating a Bootable CD

Like most things PC, the mechanism for bootable CDs is a hack. While most PCs aren't all that good at booting from CD-ROM drives, they are really good at booting from floppies, so that's what they do. To make a CD bootable, I would first produce a floppy disk that boots the way I want it to. For instance, I would use a script similar to the following:

```
superformat /dev/fd0
syslinux /dev/fd0
mount -t msdos /dev/fd0 /mnt
cp vmlinuz /mnt
echo "prompt 1" > /mnt/syslinux.cfg
echo "timeout 100" >> /mnt/syslinux.cfg
echo "default vmlinuz" \
```

```
>> /mnt/syslinux.cfg
echo "append root=/dev/hdc" \
>> /mnt/syslinux.cfg
umount /mnt
```

Now, I would reboot my test machine using the floppy to see that it loads the kernel properly and mounts the correct root device (/dev/hdc in this example). If all seems well, then I would make an image of the floppy with dd using a command like:

```
# dd if=/dev/fd0 of=boot.img
```

The file boot.img should then be placed within the directory tree from which the ISO file system image will be created. Finally, the -b option of mkisofs will cause it to place a special entry in the ISO9660 catalog pointing to this floppy image file. What the PC's BIOS actually will do at boot time is map this file on the CD to an 'emulated floppy' and then try to boot from it as if it were a real floppy disk. The rest of the BIOS and any boot loader residing on the floppy image are usually completely fooled. Of course, once the Linux kernel is loaded, the whole illusion evaporates and the CD-ROM goes back to being an ordinary device like /dev/hdc. The following commands would be typical for creating the actual ISO image using mkisofs:

```
# cp boot.img CD/boot/boot.img
# mkisofs -v -R -b boot/boot.img \
    -c boot/boot.catalog \
    -o img/lpacd.iso CD
```

Note that the path given in the -b option should be the path relative to the root of the CD file system. At this point the only remaining task would be using cdrecord to actually write the ISO image to a blank recordable CD:

```
# cdrecord -v speed=4 dev=2,0,0 \
    img/lpacd.iso
```

This command is typical of what I would use on my own machine, but the speed= and dev= parameters are specific to the particular hardware in use. The -scanbus option to cdrecord will show what devices are available and what value to pass in the dev= parameter for each particular drive. The speed= parameter should be set to a value compatible with the capabilities of both the drive and media used. The mkisofs utility has a number of other options for specifying boot disk and kernel images when creating bootable images. Although the LxA example configurations have not yet been built for Sparc hardware, bootable ISO images for that platform can be produced as well. For more comprehensive details regarding the usage of mkisofs and cdrecord [13, 7].

One important choice that must be made when creating boot and root disks on recordable CD-ROM media is whether to use CD-R (write-once) or CD-RW (re-writable). Despite the obvious penalties for mistakes when using a write-once media, I prefer to use CD-R unless I am completely sure that all target machines have CD-RW compatible drives installed.

Many older CD-ROM drives are not capable of reading or booting from CD-RW disks, but if the drive includes 'Multi-Read' compatibility in its specifications it should read either media with no problem.

Using the Kernel Command Line

In order to allow the maximum amount of flexibility for booting an LPA-CD image without requiring changes to the rc.init script burned onto the CD, I decided to employ a couple of custom kernel command line parameters to pass information to rc.init through the boot loader. The basic idea when using the kernel command line on Linux is to insert whatever parameters are required into the append= parameter of the boot loader and then take advantage of the Linux /proc/cmdline node to extract these parameters from the kernel at runtime. For the sake of LPA-CD, two parameters were required, the block device and the filesystem type for the /local filesystem. The command line parameters are read from the file syslinux.cfg by the syslinux boot loader at startup time and are specified as follows:

```
append root=/dev/hdc lxa_lfs=vfat \
        lxa_ldev=/dev/fdl
```

These parameters are then passed on to the kernel by the boot loader and may be retrieved from the /proc/cmdline node with various utilities such as awk. The actual rc.init script used for LPA-CD and illustrating this technique is shown in Listing 3. Any number of utilities such as cut, sed, or grep, or even perl can be used for this sort of task, but I chose awk since it was the smallest binary which provided a general purpose scripting and text processing tool available on my development system.

Example LxA Configurations

LRA: Linux Routing Appliance

This system provides: a DHCP server, local and caching DNS, dial-on-demand PPP, firewalling, and

masquerading for a local network. The entire system boots from a single floppy and runs from a RAM disk. The minimum hardware requirements are: a 486 CPU, 16MB RAM, an ethernet card, a 3.5" floppy, and a modem. LRA was the first configuration developed and seems from analysis of the LxA site access statistics to be the most popular among users.

LFA: Linux Firewall Appliance

LFA requires little more than a floppy disk drive, an ethernet card, a CPU, and some RAM. Users of LFA have reported that it works adequately with as little as 6MB of RAM even with the interactive login package added. All of the work of this system is done by the kernel's own firewalling and routing modules, and user reports suggest that LFA can function well for moderate traffic loads across one or two 10Mbps ethernet segments even with only a 486 CPU. Heavy loads, more segments, or 100Mbps ethernet would definitely require at least 16MB of RAM and a Pentium class CPU for satisfactory operation. LFA is, I believe, almost as minimal a Linux system as one could possibly assemble and still perform useful work. After the startup scripts have run the only processes besides the kernel and init on this configuration are syslogd and klogd. Because of the minimal process load, LFA is extremely stable, runs entirely from a RAM disk, and leaves little opportunity for compromise or failure. This system can provide firewalling, routing, and masquerading between two or more ethernet networks. For many small businesses or isolated departments, a commercial hardware or software firewall package may be excessively costly and present many features that would go unused. In situations such as these, LFA can provide a secure and reliable alternative which supports many of the most useful features of a commercial firewall or router at much lower cost. Excluding the /dev directory, the root filesystem for this configuration contains only around forty files and directories; see Table 1.

```
#!/bin/sh
PATH=/bin:/sbin
export PATH

# Mount /proc filesystem first
mount -n -t proc proc /proc

# Parse custom command line parameters from /proc/cmdline
eval `cat /proc/cmdline | awk '{ match($0, /lxa_lfs=[^ ]+/); \
    printf("LXA_LFS=%s ; export LXA_LFS;\n", \
    substr($0, (RSTART + 8), (RLENGTH - 8))); }'`
eval `cat /proc/cmdline | awk '{ match($0, /lxa_ldev=[^ ]+/); \
    printf("LXA_LDEV=%s ; export LXA_LDEV;\n", \
    substr($0, (RSTART + 9), (RLENGTH - 9))); }'`

# Mount /local filesystem
echo -n 'Mounting '$LXA_LDEV' on /local as type '$LXA_LFS'...'
mount -n -t $LXA_LFS $LXA_LDEV /local
echo 'done.'
exit 0
```

Listing 3: The rc.init script used for LPA-CD.

Path	Comment
/bin	location for user binaries
/bin/sh	ash, an interpreter for startup and other scripts
/bin/hostname	sets and echos the system hostname
/bin/echo	used for writing configuration settings into /proc filesystem nodes and displaying boot messages
/etc	location for system configuration and boot files
/etc/inittab	configuration file for /sbin/init
/etc/fstab	list of active filesystems, their locations, and options
/etc/hostname	contains the system hostname
/etc/syslog.conf	configuration file for /sbin/syslogd
/etc/hosts	local hostname to IP address mappings
/etc/rc.init	first setup script executed by /sbin/init
/etc/rc.local	main system startup script
/etc/networks	local network name to IP subnet mappings
/etc/services	mapping of port numbers to service names
/etc/protocols	mapping of protocol id numbers to protocol names
/etc/passwd	list of local user accounts
/etc/group	list of local user groups
/etc/nsswitch.conf	configuration file for GNU C library map functions such as gethostbyname()
/lib	location for dynamically linked libraries
/lib/ld-linux.so.2	the dynamic library loader, used by all dynamically linked executables
/lib/libc.so.6	the standard GNU C runtime library, huge but essential
/lib/libnss_files.so.2	name service switch module used by C library for lookups from files such as /etc/hosts
/lib/modules	location for kernel modules
/lib/modules/*	any required drivers for devices such as ethernet adapters
/linuxrc	optional pre-init script executed by the kernel at startup (not used by LxA)
/proc	mount point for the kernel's proc pseudo filesystem
/sbin	location for system binaries
/sbin/hwclock	used to sync system time with hardware clock
/sbin/ifconfig	configure network interfaces
/sbin/init	process id 1 started by the kernel at boot time
/sbin/ipchains	used to configure the kernel's IP firewalling rules
/sbin/mount	used to mount and remount /, /proc, and any other filesystems
/sbin/route	used to configure network routing tables
/sbin/insmod	used to load modules such as ethernet drivers
/sbin/syslogd	system message logging daemon
/sbin/klogd	kernel message logging daemon
/tmp	location for working and scratch files
/var	location for spool and log files
/var/run	location for lock and pid files

Table 1: Root filesystem.

Due to its extremely minimalistic nature, I have also found LFA useful as a baseline configuration for creating things like custom rescue and repair diskettes.

LPA-CD: Linux Printing Appliance on CD-ROM

System Requirements

LPA-CD is a complete Linux system which will boot and run directly from the ISO image provided on the LxA project site. The minimal system requirements for an LPA-CD system are: a 386 or 486 CPU, 4MB RAM, an ethernet card, a floppy disk drive, a CD-ROM drive, and obviously at least one printer. Recommended but optional items include a fixed disk or Zip drive for temporary files and swap space and as much RAM as the budget will allow for better performance and higher load capability. While LPA-CD contains many more capabilities than LFA, it benefits slightly in terms of RAM requirements as a result of mounting its root file system from the CD-ROM instead of a RAM disk. The minimal configuration has proven to work well for a light load such as a dozen Windows clients printing small to moderately sized documents through client side PCL drivers to PCL printers. Situations involving large numbers of clients, large print jobs, or conversion of complex, color PostScript jobs will benefit from as much CPU, RAM, swap, and spooling space as the system can hold. Only evaluation of the anticipated system load can dictate what hardware configuration can be considered minimal for a given situation.

Features

An LPA-CD system will spool, convert, and output print jobs from either UNIX lpr or Windows Samba clients. This package allows easy setup and management of print spooling hosts for mixed platform networks. LPA-CD is not based on any particular distribution, but the original sample configuration included components from the Debian 2.2 (Potato) system with the Linux kernel version 2.2.14 and glibc version 2.1.3. In the current release, LPA-CD includes the option for a minimal but complete set of interactive tools and a sulogin shell for use during staging. Since the system includes tools like vi, it is possible to save configuration changes to onto the local configuration floppy while the system is operating (something that is not possible with many of the more resource restricted configurations such as LRA, LFA, and LPA). This allows an LPA-CD box to be started and configured completely standalone if desired. Basic features in the current release of LPA-CD include:

- Linux kernel version 2.2.17
- Glibc 2.1.3
- Linux ipchains kernel firewalling
- PAM (Pluggable Authentication Modules)
- Samba for serving Windows printing clients
- BSD lpr for UNIX printing clients
- Ghostscript for conversion of PostScript/PDF print jobs to printer-native output

- Netpbm for conversion of various graphics formats to printer native output
- Magicfilter for easy and flexible configuration of print job conversion based on file magic numbers

LPA-CD uses the standard BSD lpd for spooling of print jobs, and can support both directly connected serial or parallel printers as well as network attached devices such as HP JetDirect adapters. LPA-CD fills a unique application niche since it provides both print job spooling and automatic conversion of many job formats for UNIX as well as Windows clients. The TODO list for LPA-CD also includes adding support for MacOS clients via the Netatalk package in the near future.

System Configuration

An LPA-CD system is comprised of two disk images, a bootable CD-ROM and a bootable floppy. Either may be used to bootstrap the target machine since many older machines don't have a BIOS capable of booting from a CD-ROM drive. The result is the same in either case; the root filesystem will be the ISO9660 filesystem found on the CD, and all configuration files that would need to be changed locally reside on the floppy image which will be mounted on /local at system startup. LPA-CD offers two choices for a quick start; using disk images or using a tarball. If the environment will require only changing some configuration files, then the disk images are probably easier to start with. If, however, extensive changes or additions to the CD image are anticipated, then the tarball will prove more suitable. The CD image and bootstrap procedures are intended to work well for the majority of typical setups, and the tarball is probably only appropriate if the configuration requires unusual features or extensive changes. In either case, of course, either a CD-R or CD-RW drive is required for actually writing the ISO image to a CD-R disk.

System Startup

The rc.init script included on the CD image is responsible for mounting the appropriate filesystem on /local, and the /local/rc.local script actually does most of the work of starting up an LPA-CD machine. This script should be tuned for the specific machine on which it will be run. Only a summary description of the script is provided here. The phases of startup include:

- Loading modules: Near the top of the file are several invocations of modprobe which load driver modules for devices such as the ethernet adapter.
- Preparing and activating the /var filesystem: After any required modules are loaded, there are several alternative sections in the rc.local script showing examples of how to create and/or mount various types of filesystems for use as temp and spooling areas. Only one of these alternatives should be used, and the others

may be removed or ignored. As mentioned in the Issues section, a careful evaluation of the resources available and the anticipated load should be made before choosing this option. Examples for each alternative are shown here:

```
# RAMDISK
dd if=/dev/zero of=/dev/ram0 \
    bs=1024 count=4096
mkfs.minix -v -i 8192 /dev/ram0
mount -t minix /dev/ram0 /var

# LOOPBACK IMAGE
losetup /dev/loop0 /local/var.img
mkfs.minix -v -i 8192 /dev/loop0
mount -t minix /dev/loop0 /var

# LOCAL DISK
mkfs.minix -v -i 8192 /dev/hdb1
mount -t minix /dev/hdb1 /var

# NFS
modprobe nfs
mount -t nfs \
    spoolhost:/spool/linprinter /var

# Unpacking /var filesystem image
tar -C /var -xvzf \
    /local/var.tar.gz
```

The -v option to mkfs.minix causes a Minix V2 filesystem to be created which is necessary for volumes over 64MB in size. The -i 8192 option causes allocation of 8192 inodes in the new filesystem which is a somewhat larger number than the default. If volumes of more than 256MB will be used, then the ext2 filesystem is probably a better choice.

- Activating network interfaces: This section includes standard ifconfig and route invocations to set up the interfaces and routes as necessary for the particular environment.
- Configuring ipchains rulesets: The default ipchains statements included install a fairly strict set of filters allowing only the traffic necessary for lpr and Samba clients to connect. The rules also allow clients on the local network to ping and be pinged by the LPA-CD box to aid in diagnostics.
- Starting services: This section at the end starts syslogd, klogd, nmbd, smbd, and lpd.

Sulogin Shell

The CD image contains a minimal but mostly complete set of tools for system setup, monitoring, and diagnosis.⁵

⁵In production configurations many of these tools may go unused, but they are included in the ISO image for LPA-CD by default since the cost to add them is relatively much higher for a write-once media like CD-R versus a re-writable media such as a floppy. The floppy based configurations such as LRA include in their configuration scripts the option to add a login shell if desired.

The only non-standard tool included with LPA-CD is cryptwd which is a small utility designed to help replace the ordinary

- Interactive/miscellaneous commands: awk bash clear cryptwd echo sh vi
- File operations: cat cp chmod chown dd find grep less ln ls mkdir mv rm touch
- Filesystem maintenance: df dosfsck e2fsck fdisk fsck fsck.ext2 fsck.minix fsck.msos fsck.vfat losetup mkdosfs mke2fs mkfs mkfs.ext2 mkfs.minix mkfs.msos mkfs.vfat mkswap mount swapoff swapon umount
- File archive maintenance: cpio gzip tar
- System status: date dmesg free hostname lsmof lsof ps strace uname
- System/process Control: halt hwclock insmod kill modprobe reboot rmod setserial shutdown tunelp
- Networking: ifconfig ipchains netstat ping route
- Samba: mksmbpasswd smbclient smbpasswd smbpool smbstatus testparm
- BSD print spooling: lpc lpq lpr lprm lptest pac
- Print job file format conversion: magicfilter bmp-toppm djpeg dvips fig2dev giftopnm gs pngtopnm pnmtops rasttopnm sgitopnm tiff2ps

Conclusions

Open source operating systems present unique opportunities for creation of ultra light systems tuned to specific tasks. By exploiting the flexibility and rich knowledge base of systems like Linux, FreeBSD, OpenBSD, and other open source systems, administrators can greatly reduce the time required to set up, maintain, and upgrade the systems under their control. LxA facilitates this process by providing documentation, tools, and ready-to-run reference implementations for several useful Linux based network appliances. By taking a different approach to composing its Linux based platform, LxA reduces system complexity while increasing security and reliability. In addition, LxA will hopefully be able to quickly incorporate new versions of critical system components when available due to its use of standard components wherever possible. When compared to commercial UNIX systems, Linux presents more options for the average system administrator in building unusual or small system configurations due primarily to the following features:

passwd tool. As discussed previously, the standard passwd utility will not function correctly with a read-only /etc directory, and this obviously causes problems for a CD based root filesystem. The cryptwd tool (in combination with vi) will allow the changing of user account passwords on a standalone LPA-CD system. To use cryptwd, one would first open the /etc/passwd file in vi and then use an ex command like the following: :r ! cryptwd <newpassword>. This will place a string representing the crypt version of the password specified into the current editing session. If it is necessary to include special characters in the password string, they may be escaped with a '\ ' like this: :r ! cryptwd \%&chevy. Of course, this command can also be used at a shell prompt if the crypt version of some string is needed. This utility chooses a crypt salt for the password randomly, so there is no need to specify one on the command line.

- A standard and flexible RAM disk device
- The initial RAM disk boot capability
- Ease of composing minimal systems suitable for running from removable media
- Generally more sophisticated tracing tools
- Free access to source code for troubleshooting and modifications where necessary
- An overall system ethos focused on flexibility and hackability

The LxA project has definitely succeeded in providing me with a more thorough understanding of the components involved in bootstrapping and running Linux and other UNIX-like systems. In more practical terms, LxA configurations have made it feasible for me to continue supporting customers located at great geographic distances from my new home without excessive travel or expense. As with any open source project, it is difficult to estimate accurately any rate of deployment or success for LxA outside the realm of my own experience. I can say, however, that there have been approximately twenty to thirty thousand visits to the LxA project site since it was first created around December of 1999. Of those who have visited, at least several thousand have taken the time to download one or more of the sample configurations. Of the several dozen users who have sent me email regarding LxA, all have reported positive results and at least several have suggested new features and configurations which they would find useful. I have received at least a dozen replies to a request for usage reports so far, with those responding most often finding a use for the LRA and LPA-CD configurations. Purely subjective observation suggests that a large proportion of active LxA users are from outside the U.S., and I would suppose that this can be attributed to the emphasis the project places on its relatively low resource requirements and the prevalence of older hardware platforms in many areas of the world.

Future Work

TODO List

- More elegant make scripts and better support for RedHat based host distributions.
- Analysis scripts to automate the process of identifying and collecting all the components and dependencies of new services for LxA configurations
- A simple console dialog and/or X based GUI configuration utility for building LxA root file system trees and disk images.
- Enhancement of the LPA-CD root file system image with additional components to form a more general-purpose 'LxA Platform CD'. This will include sshd for remote administration as well as the netatalk package for support of MacOS printing clients and some other services such as dhcpd and pppd.
- Integration of the Linux kernel 2.4 when released.

Alternative Configurations

- LLA (Linux Lynx Appliance): Useful for turning ancient 386 class machines into text based POS terminals, etc.
- LMA (Linux Mozilla Appliance): Another CD based configuration; the classic thin client for creating web and Java based network application systems.
- LCDA (Linux CD-ROM Tower Appliance)
- LTSA (Linux Terminal Server Appliance)
- LVA (Linux VPN Appliance)

Availability

All of the disk images, documentation, and other materials related to LxA are publicly available via the Internet [12]. All software associated with LxA is covered, as are the components on which it is built, by the GNU General Public License [6]. Everything documented in this paper is freely available for any use as long as the distribution requirements of the GPL are followed.

Errata

Any corrections or additions to this paper will be posted on the LxA project site [12] where the full text of this paper will be posted and maintained in HTML format as well.

Author Information

Michael W. Shaffer has worked as a system administrator, software developer, and system engineer for a variety of companies and has been using Linux since he first discovered it in mid-1993. He studied Spanish, English Literature, Philosophy, and most recently Computer Science at the University of South Carolina in Columbia, SC. Although he was born in Connecticut and grew up in South Carolina, he has recently moved to Silicon Valley and currently works as Hostmaster, Postmaster, and Security Officer for the Research Computing Services department of Agilent Laboratories in Palo Alto, CA. Reach him at Agilent Labs RCS; 3500 Deer Creek Road; Palo Alto, CA 94304; USA. His phone number is +1 650-485.2955. His email address is: shaffer@labs.agilent.com.

References

- [1] Bastille Linux Project; <http://www.bastille-linux.org/>.
- [2] BusyBox; <http://busybox.lineo.com/>.
- [3] Embedded Linux Stargate; <http://linux-embedded.com/>.
- [4] Floppix; <http://floppix.ccai.com/faq.html>.
- [5] Gibraltar Firewall Project; <http://www.gibraltar.at>.
- [6] GNU Project General Public License; <http://www.gnu.org/copyleft/gpl.html>.
- [7] Linux CD Writing HOWTO; <http://www.linux.org/help/ldp/howto/CD-Writing-HOWTO.html>.

- [8] Linux Kernel HOWTO; <http://www.linux.org/help/ldp/howto/Kernel-HOWTO.html> .
- [9] Linux kernel source tree documentation (including initrd.txt and other files in the Documentation directory); <http://www.kernel.org> .
- [10] LOAF; <http://loaf.ecks.org/> .
- [11] LRP (Linux Router Project); <http://www.linuxrouter.org/> .
- [12] LxA Project Homepage; <http://equusasinus.com/lxa/index.html> .
- [13] Mkisofs and cdrecord documentation (including cdrecord(1), mkisofs(8), and README.eltorito); <http://www.fokus.gmd.de/research/cc/gclone/employees/joerg.schilling/private/mkisofs.html> .
- [14] OpenBSD Project; <http://www.openbsd.org> .
- [15] Open Directory Tiny Linux Page; http://dmoz.org/Computers/Software/Operating_Systems/Linux/Distributions/Tiny_Linux/ .
- [16] TAPR CompactFlash/IDE adapter; <http://www.tapr.org/tapr/html/Fcfa.html> .
- [17] Trinux; <http://www.trinux.org/> .
- [18] Yahoo! Linux Distributions Page; http://dir.yahoo.com/Computers_and_Internet/Software/Operating_Systems/Unix/Linux/Distributions/ .

Automating Dual Boot (Linux and NT) Installations

Rajeev Agrawala, Rob Fulmer, & Shaun Erickson – Lucent/Bell-Labs Research

ABSTRACT

This paper presents a solution for automating dual boot Linux and Windows NT 4.0 installations on a PC. The process is automated to the extent that one starts with a PC with a blank hard disk and ends up with a fully customized dual-boot PC. The process involves booting a PC with a floppy disk; choosing what kind of system is required from a menu and then, once the install begins, popping out the disk. The unattended PC goes through installing and customizing the Linux installation. It then automatically boots into DOS and installs Windows NT 4.0. The whole process takes about an hour or two, depending upon the type of installation and network speed. After the process is complete, the PC is ready, fully customized to the user's local environment. The solution presented in this paper can be used to do either Linux-only or NT-only installs in addition to dual-boot configurations.

Introduction

Automating the operating system's installation has always been a topic of interest to System Administrators. In the era of installing Windows 3.1 and applications, automation meant eliminating the process of feeding 30 floppy disks one by one, only to find out that the 27th disk had a bad sector error. Solaris provides JumpStart, with which one can automate OS installation. IRIX, in its latest release (6.5), provides Roboinstall to automate the OS installation. We have automated Windows NT 4.0 installs [1]. Linux (Redhat) provides Kickstart to automate the OS installation [2].

All of these methods automate the process of one OS installation only. Our environment initially consisted of Unix workstations on desktops. Then the trend shifted towards PCs running Windows NT 4.0 on desktops and Unix servers for the backend processing. With the popularity of Linux, more and more users in our environment started asking for PCs running both Linux and NT with a choice to boot in either operating system. With this trend, the support group had to spend a lot of time on each PC to make it dual boot. Also, each installed PC was one of a kind, depending upon the way a particular administrator decided to customize and install it. The method described in this paper was developed so those system administrators could install PCs more efficiently and consistently.

Overview

The installation process uses a modified "boot-net" disk from RedHat Linux Kickstart. The PC is booted off this disk, and a menu is presented to the administrator. The menu typically consists of the choices for different ways of customizing the installation. The choices, for example, could be

- Both Linux and NT install customized for department A, or B, or C.

- Only Linux install for department A, or B, or C
- Only NT install for department A, or B, or C
- Generic Linux and NT install (No customization)
- Generic Linux install
- Generic NT install

Once the system administrator chooses the installation type from the menu, the installation starts. At this stage, he can pop the floppy disk out and walk away from the PC. From this point onwards, the installation goes unattended. When the installation is finished, the PC is ready with both operating systems and is usable. Since Redhat Linux does not support the latest video cards out of the box, we skip the installation of X altogether for consistency. Also, sound installation is not supported through kickstart. Therefore, with our current setup, one has to configure X and audio after the installation is done.

The steps involved in doing a dual boot installation are:

- Starting the first OS installation
- Customization of first operating system
- Preparing/repartitioning and formatting disk for second OS installation.
- Copying the second OS installation files to the partition prepared in previous step.
- Handing over control to second operating system install program.
- Second operating system installation and customization.
- Passing control back to first operating system.

The first step above is started manually by the administrator. The rest of it happens automatically.

Constraints

The seven steps mentioned above impose some restrictions on what can be chosen as the first operating system and what can be second.

The following problems need to be addressed to be able to do a successful install:

1. The installation mechanism should be capable of installing either operating system or both.
2. There is some customization required, based on departments, user requirements, etc. Therefore, it should be possible to specify the type of customization for both operating systems at the start of the installation process.

3. The first operating system should have knowledge about the filesystem for the second operating system and should be able to make the file system bootable for it.

We shall see later in this paper how the first two requirements are met. The third requirement makes Linux a natural choice for the first operating system, since Linux can create a DOS filesystem through

```

Default deptA
Prompt 1
Display dualboot.msg
F1 dualboot.msg
F2 other.msg
F3 expert.msg
F4 param.msg
F5 rescue.msg
F6 boot.msg
F7 snake.msg
Label linux
    Kernel vmlinuz
    Append initrd=initrd.img network
Label text
    Kernel vmlinuz
    Append initrd=initrd.img network text
Label expert
    Kernel vmlinuz
    Append expert initrd=initrd.img network
Label ks
    Kernel vmlinuz
    Append ks initrd=initrd.img network
Label deptA
    Kernel vmlinuz
    Append initrd=initrd.img network ks=nfs:serverIPAddress:/kickstart/ks.cfg profile=deptA
Label 1
    Kernel vmlinuz
    Append initrd=initrd.img network ks=nfs:serverIPAddress:/kickstart/ks.cfg profile=deptA
Label generic
    Kernel vmlinuz
    Append initrd=initrd.img network ks=nfs:serverIPAddress:/kickstart/ks.cfg profile=generic
Label 2
    Kernel vmlinuz
    Append initrd=initrd.img network ks=nfs:serverIPAddress:/kickstart/ks.cfg profile=generic
Label ntonly
    Kernel vmlinuz
    Append initrd=initrd.img network ks=nfs:serverIPAddress:/kickstart/ntonlyks.cfg profile=ntonly
Label 3
    Kernel vmlinuz
    Append initrd=initrd.img network ks=nfs:serverIPAddress:/kickstart/ntonlyks.cfg profile=ntonly
Label deptAlinux
    Kernel vmlinuz
    Append initrd=initrd.img network ks=nfs:serverIPAddress:/kickstart/linux-server.cfg profile=deptAlinux
Label 4
    Kernel vmlinuz
    Append initrd=initrd.img network ks=nfs:serverIPAddress:/kickstart/linux-server.cfg profile=deptAlinux
Label genericlinux
    Kernel vmlinuz
    Append initrd=initrd.img network ks=nfs:serverIPAddress:/kickstart/linux-server.cfg profile=genericlinux
Label 5
    Kernel vmlinuz
    Append initrd=initrd.img network ks=nfs:serverIPAddress:/kickstart/mathsummer.cfg profile=genericlinux

```

Figure 1: Modified SYSLINUX.CFG file.

available tools, and, with some effort, it is possible to make the DOS partition bootable.

The Details

First we place the Linux distribution on a Unix server. This can be downloaded from the RedHat Web site at <http://www.redhat.com>. This distribution should be exported read only through NFS.

Boot Floppy

We next create a boot floppy from the bootnet floppy image provided by the RedHat Linux distribution. We then modify the `syslinux.cfg` and create a file named `dualboot.msg` to include the menu to be presented to the administrator. Figure 1 shows part of modified `syslinux.cfg` file for the possible menu choices mentioned earlier. Figure 2 shows the `dualboot.msg` file, which is presented to the system administrator at startup.

Let's analyze the line shown in bold in Figure 1. This line is used by `syslinux` to invoke the kernel, when the administrator chooses 'dual boot install for department A'. This line tells the kernel to use kickstart to do the Linux installation and use the file `ks.cfg` from `serverIPAddress` as the kickstart configuration file. Here the administrator will need to replace the "serverIPAddress" with the IP address of the NFS Server where the `ks.cfg` file is stored. This line also specifies 'profile=deptA'. This is not a kickstart or kernel directive. The kernel is passed this parameter profile but ignores it. Later in the postinstall section, we retrieve this parameter from the kernel and use it to customize Linux and NT. `Syslinux` specifies two configuration files. One is the kickstart configuration file, used by kickstart. The second is a customization configuration file, which is used by the configuration control program `ksconfig`. `Ksconfig` is a perl script, which is the heart of the installation system. It takes control of the installation during the %post section of kickstart and then drives the rest of the installation process using the directives in the profile parameter.

Figure 2 shows the 'DUALBOOT.MSG' file. This file is displayed when `syslinux` starts from the boot floppy. The text in the file acts as the menu text for the administrator.

Note that by passing the profile parameter to the kernel, we solve the first constraint mentioned in earlier. As we will see later, the profile specifies the customization for both the operating systems.

Kickstart

We encourage the reader to read about kickstart in reference [2]. Here we give a brief description of kickstart, as it relates to this paper.

Figure 3 gives a typical `ks.cfg` file for used for Linux installation. Here we provide the anatomy of important lines and their significance. For the sake of clarity, we have added line numbers, but they are not part of the actual `ks.cfg` file.

Line 3 instructs kickstart to use network mode and use DHCP to get IP information.

Line 5 specifies `/linux/redhat-6.2` as the distribution directory using NFS from kickstartServer.

Line 9 says to restore the standard MBR to the disk. This option didn't work for us for some reason, so we had to do it later in the customization section.

Line 11 instructs kickstart to delete all the partitions on the disk.

Line 13 creates a partition of type `ext2` which will be mounted as `/boot`. The `grow` parameter on this line causes the partition to grow to a maximum. In reality, this will be constrained, as we will see later.

Line 14 creates a swap partition of size 256 MB.

Line 15 creates an `ext2` type partition, which will be mounted as `/`. The `grow` parameter on this line causes it to grow to the maximum available disk space.

Since Line 13 and Line 15 both specify the `grow` option, both the partitions compete for available disk space and get roughly half of the total disk space, minus the swap partition. Kickstart also restricts the `/boot` partition to be a maximum of 1024 cylinders (due to boot restrictions of LILO), so the `/boot` partition gets a maximum of 8GB disk if the disk is larger than 16GB. Also, as of the writing of this paper, Kickstart will create a primary partition for `/boot` and put swap and `/` in an extended partition.

Line 17 sets the kickstart mode to install rather than upgrade.

```

Welcome to Red Hat Linux 6.1!

Please choose from the following options
 1. deptA      (Dual boot install linux customized for deptA)
 2. generic    (Dual boot install linux not customized)
                (use deliver script before delivery)
 3. ntonly     (Finally only NT is left installed on the disk)
 4. genericlo  (only Linux installation - uncustomized)
You can still type commands like ks profile=<someprofile> or
Use other kickstart config file by specifying in DHCP configuration.
You can still use other linux boot floppy commands.
[F1-Main] [F2-Other] [F6-Linux Boot Screen]
```

Figure 2: DUALBOOT.MSG file.

Line 23 instructs kickstart to skip the X server installation. If we don't skip this, kickstart will fail for an unsupported video card.

Line 25 sets the root password for the machine.

Line 29 instructs LILO to install its boot sector on the /boot partition instead of in the MBR.

Line 31 sets up kickstart to reboot automatically, instead of waiting for the user to press return.

Line 34 specifies installation of the complete RedHat distribution.

In the %post section, Line 38 NFS mounts the directory which contains the ksconfig file.

Line 39 executes the script ksconfig. From this point onwards, the ksconfig script takes control of the

installation. The complete functionality of this script is discussed later.

Ksconfig Script

During the %post section of kickstart, the ksconfig script is executed. Before the system's first reboot, this script installs itself to be run during runlevels 1 and 3. This script also changes the /etc/inittab file to set the default runlevel to 1, so once kickstart is done and the machine reboots, it goes into runlevel 1.

The ksconfig script is mostly profile driven. It extracts the name of the profile, which was passed to the kernel by syslinux at the start of installation. Although Linux does not provide any direct way of obtaining the invocation time parameters, it keeps them in the /proc/cmdline file. The ksconfig script uses

```

1. lang en_US
2. #
3. network --bootproto dhcp
4. #
5. nfs --server kickstartServer --dir /linux/redhat-6.2
6. #
7. keyboard us
8. #
9. zerombr yes
10. #
11. clearpart -all
12. #
13. part /boot --size 1 -grow
14. part swap --size 256
15. part / --size 1 -grow
16. #
17. install
18. #
19. mouse logimmanps/2
20. #
21. timezone US/Eastern
22. #
23. skipx
24. #
25. rootpw !#%&wtqr
26. #
27. auth --useshadow --enablemd5
28. #
29. lilo --location partition
30. #
31. reboot
32. #
33. %packages
34. @ Everything.
35. #
36. %post
37. /bin/mkdir /mnt2
38. /bin/mount serverIPAddress:/kickstart /mnt2
39. /mnt2/ksconfig
40. /bin/umount /mnt2
41. /bin/rmdir /mnt2

```

Figure 3: A typical kickstart configuration file (ks.cfg) used for Linux installation.

this file to get the profile name. The script assumes that all the profiles are available in the directory from which it was run. It copies the profile to the local system. A sample profile is given in Figure 4. This profile is divided into sections. Each section header is enclosed within '%' characters.

The format of the section header is given below:

```
%SectionName runlevel-validpasses%
```

SectionName is the name of the section. The valid section names are:

- Environment
- Disklayout
- Nt
- Linux
- Installapplications
- Prepnt
- Config
- Zerombr
- Removepackages
- Cleanup

The section names are case insensitive.

Runlevel-validpass is an optional control string. If present, runlevel is the Linux runlevel, during which this section is active. So, with a section header specifying a runlevel of 3, it will be processed during

runlevel 3 only. During the %post section of kickstart (before the machine is rebooted), the runlevel is not set. To process a section during the %post section itself, runlevel should be specified as 'i' (signifying install runlevel).

Validpass is a combination of the following values:

- 0-9 – A number specifying the installation pass during a particular runlevel (e.g., 1 means first reboot in the specified runlevel, 2 means 2nd reboot in the same run level etc.).
- 'd' – The installation pass is set to 'd' when the ksconfig script is manually run to customize a machine after a generic installation. The letter 'd' signifies delivery pass (when the PC is to be finally delivered to the user).

For example, if the administrator decides to process a section during runlevel 1 and pass 2, he will specify 1-2 in the section header. When the ksconfig script first boots into runlevel 1, it sets the pass number to 1. Since the section is valid in pass 2, it will reboot the machine once again in run level 1 (after processing other applicable sections), increase the pass to 2 and then process the section. Specifying '1-1d' as the control will execute the section after first reboot in

<pre>%environment% logFile=/var/log/ksconfiglog dualboot=true %endEnvironment% %diskLayout 1-1% #saves /boot partition on root partrootdev1=saveOnPartrootdev6 #unmounts /boot partition partrootdev1=umount #deletes /boot partition partrootdev1=delete #Creates a 30M ext2 partition starting from cylinder 1 partrootdev1=30M,ext2,1 #Makes e2fs filesystem partrootdev1=mke2fs #Marks this partition active partrootdev1=markactive #mounts it on /boot partrootdev1=mouton /boot #restores the original /boot contents partrootdev1=restoreFromPartrootdev6 #create a 2047M DOS Partition after /boot partition partrootdev3=2047M,FAT16,afterpartrootdev1 %endDiskLayout% %nt% %diskLayout 3-1% partrootdev3=mkdosfs %endDiskLayout% %installapplications 3-1% acrobat audix exceed gsview mcafee netscape office97 reskit</pre>	<pre>security ssh winzip %endInstallApplications% %prepNt 3-1% hostname=useDummy preparent=rootdev3 %endPrepNt% %endnt% %linux% %zerombr i-1% rootdev %endZerombr% %removePackages 1-1d% rpmRemCmd=/bin/rpm --quiet --allmatches --nodeps -e cxhextris gnome-games gnome-games-devel gnuchess howto-chinese kdegames * * * %endRemovePackages% %config 1-1% copyFile=functions customFile=deptA.1-1 %endConfig% %config 3-1d% customFile=deptA.3-1 %endConfig% %cleanup 3-1d% /save /etc/rc.d/init.d/ksconfig %endLinux%</pre>
---	--

Figure 4: A sample profile 'deptA' for ksconfig.

runlevel 1, and again if the profile is used to manually customize a generic machine.

To better understand the use of pass 'd', consider a scenario where PCs are ordered in bulk for various departments and kept preinstalled. Since, at the time of installation, it is not known which department a particular PC will go to, all the PCs are initially installed with no customization. At the time of delivery, the `ksconfig` script can be run manually with the profile name for the particular department. At that time only sections marked with pass 'd' will be executed. So the same profile can be used either at the time of customized installation from the start or during manual customization.

The `%endSectionName%` directive marks the end of a section. Sections can be nested.

Let's now look at the sample profile 'deptA' given in Figure 4. The profile is read top to bottom by the `ksconfig` script. Any sections not valid for the current runlevel and pass are skipped.

Now we shall discuss the various sections. This discussion is not necessarily in the same order as it appears in the sample profile given, but in the sequence in which the sections are executed.

Environment Section

The first section name is 'environment'. There is no control string, so the section will be executed in all run levels and passes. The first command in this section is `logFile=filename`. `Ksconfig` will log the installation in this file. The only other command in this section is `dualboot=true`. This tells the `ksconfig` script that this machine is going to be a dual boot system. The other choices are 'linux' or 'nt'.

Zerombr Section

The `Zerombr` section is executed in runlevel 'i'. This means it is executed in the `%post` section of `kickstart`. The only command in this section is 'rootdev'. This command instructs `ksconfig` to restore the Standard MBR to the root disk. For a discussion on what is a root disk, see the next section. This section was introduced as a workaround to Linux Kickstart's `zerombr` [2], as the command actually did not restore the standard MBR (as of RedHat 6.1). To do this, the `ksconfig` script `dd`'s (the Unix `dd` command) 440 bytes of standard MBR to `/dev/hda` (or the root device).

DiskLayout Section

In the example profile, there are two `DiskLayout` Sections. The first section is processed in first pass of runlevel 1. This means it is done after `kickstart` is finished and the machine reboots the first time in single user mode. This section does the disk repartitioning for the installation of Windows NT. As you will recall, during `kickstart`, we created two `ext2` type partitions, roughly the same in size, occupying half of the disk each. The first partition was mounted on `/boot`, and the 2nd partition was mounted on '/'. Now we will save the contents of the `/boot` partition, delete the `/boot`

partition, create a 30M `/boot` partition and restore the `/boot` files saved earlier. In the rest of the empty space so created, we will make a 2047MB, `FAT16` partition and make a `DOS` filesystem on it. For a dual boot installation, this assumes a minimum disk of size 5 GB. This value can be tweaked, based on the standard NT load. However, if the disk is larger in size and this partition is greater than 2GB, the rest of the partition is not wasted. The NT installation grows this partition to the available size. Figure 5 shows the disk partitioning as the installation proceeds.

The First command in the 'diskLayout' section is:

```
Partrootdev1 = saveOnPartRootdev6
```

This command instructs the `ksconfig` script to find out the base root device. This would be `hda` in normal IDE based systems, `sda` in SCSI based systems and `hde` if `Ultra66` IDE card is used. The `ksconfig` script takes partition 1 of the root device and saves it as a tar file on partition 6 of the root device. As can be seen in Figure 5(b), partition 1 is the `/boot` partition and partition 6 is where '/' is mounted

The rest of the commands in this section:

- Unmount the `/boot` partition.
- Delete the partition.
- Create a 30MB `ext2` partition.
- Make an `e2fs` filesystem on this partition.
- Mark the partition active.
- Mount the partition on `/boot`
- Restore the partition contents saved earlier.
- Create a 2047MB, `FAT16` partition after the `/boot` partition.

The second `DiskLayout` section is executed in runlevel 3 and is ignored in runlevel 1. This section instructs `ksconfig` to create a `DOS` filesystem on partition 3. Partition 3 was already created as type `FAT16`, in the first `DiskLayout` section.

Removepackages Section

This section is executed in runlevel 1 and passes 1 and 'd'. This means that if this profile is used during the automatic dual boot install process, it will be processed the first time the machine boots in runlevel 1. Also, if this profile is used to manually customize a generic Linux installed system, this section will be executed. This section lists various `RPMs` which should be removed from the system. The `ksconfig` script reads one package name at a time and removes the package. By default '`rpm -quiet -e`' is used to remove the package. If other packages have dependencies on the current package, it will not be removed. One can specify a command to remove the package by the directive `rpmRemoveCommand=command`. This way, if the administrator wants, he can remove the package regardless of dependencies.

Config Section

This section facilitates the execution of any script/program to customize the system. In the

example profile, there are two Config sections. The first section is processed in runlevel 1 pass 1. The valid commands in this section are copyFile and customFile. The CopyFile command simply copies the file/directory from the NFS mounted directory to the local disk on the system. The CustomFile command copies the filename specified to local disk and executes it. Though the command execution is done in the runlevel/pass specified in the header, the actual copy operation is done during the %post section of the install itself. If networking is not available (as in single user mode), the commands can still be executed. This section is an exception, in the sense that ksconfig script does not completely ignore this section during the %post phase of the installation. For security reasons, one would like to execute a configuration script

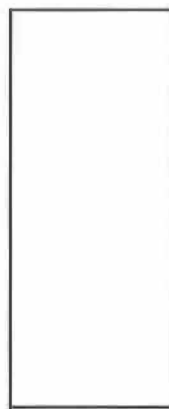
to set the root password and turn off certain services in runlevel 1, and then do the rest of the customization in runlevel 3 when all the networking is available, such as NIS, mounts etc.

InstallApplications Section

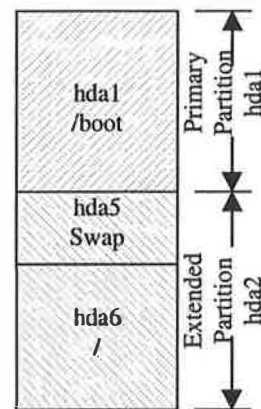
This section lists the applications that should be installed on NT. During this section the ksconfig script only makes a list of applications to be installed. This list is used during the prepNT section (discussed later), to generate a bat file (apps.bat) for NT to install these applications.

PrepNT Section

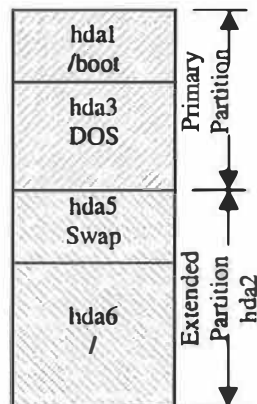
This section prepares for NT installation on the DOS partition created earlier. The only valid commands in this section are hostname and prepareNT. If



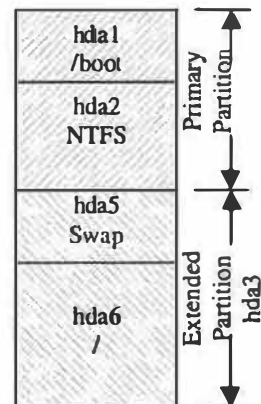
a) Empty Disk



b) Disk partition by linux install



c) Disk partition for NT install



d) Disk partition after NT install

Figure 5: Disk partition during installation process.

the hostname is specified to be 'useDummy', the script generates a dummy hostname and assigns it to the NT side. This is required because NT 4.0 does not use the hostname provided by DHCP. One of the items in my TODO list is to provide a better solution for setting the NT hostname.

The second command, prepareNT, specifies the partition that should be prepared for NT installation. A lot goes on behind the scenes to prepare the partition. A DOS filesystem was already created on the partition during the DiskLayout section.

Our automated procedure for loading NT relies on having a DOS partition pre-loaded with certain programs and data files. This is later converted into an NTFS partition. We use Linux tools to create and populate this partition. The entire process is described below:

Making Disk DOS Bootable

The first thing we need to do is to make it DOS bootable. For this, we will copy the necessary DOS files io.sys, msdos.sys and command.com to the partition. We will also copy the DOS boot sector to the first 512 bytes of the partition.

Setting Up Networking

Within Linux, we know about the Ethernet adapter, so we generate the necessary LANMAN configuration files and copy them to the DOS partition along with other LANMAN driver files for the Ethernet adapter. We keep all these files in a subdirectory under the same directory where ks.cfg (the kickstart configuration file) is kept.

Setting up to start installation

We next copy the config.sys and autoexec.bat files, which set up the MS LANMAN networking, map a directory from the network which contains the NT installation and other application installation files, and then runs the NT setup program to start NT installation. We also generate a file called apps.bat, which contains commands to install applications mentioned in 'installApplications' section (described earlier).

Passing control to DOS

After copying all the necessary files to the DOS file system, we need to pass control to DOS, to start NT installation. We do this by modifying the LILO.CONF file. LILO is set to boot into DOS by default and then the system is rebooted.

Once control is passed to DOS, the NT installation starts. The rest of the NT installation and customization process is described in detail in Autoinstall for NT [1].

Getting Control Back to Linux

During the installation, NT makes its own partition active. Therefore when the NT installation finishes, the system immediately boots NT, without going through LILO. We run a script as the last part of the NT installation to mark partition 1 active again.

This makes the boot process go through LILO, giving the choice of booting into either NT or Linux, with NT being default.

Cleanup Section

The cleanup section removes the files that were copied onto the local disk during the installation process.

Installing Other Combinations of Operating Systems

We discussed the dual boot installation process above. Along similar lines, other combinations of OSes can be installed. The SYSLINUX.CFG file in Figure 1 shows the commands to invoke kickstart for other combinations of OSes. Notice that the configuration files for kickstart and ksconfig are different for each combination of OS. For example, to start a Linux-only system, we will use a kickstart file which will only have 30 MB for boot partitions to start with, instead of using the grow option. Also in the ksconfig profile, we will remove all the NT-related sections (DiskLayout & PrepNT etc.).

To install an NT-only system, we change the Kickstart configuration file to do a minimal Linux installation instead of everything (enough for our procedures to work), and in the disk layout section, we don't create a 30MB /boot partition. Instead, we create a DOS partition starting at cylinder 1. We also delete the extended partition (containing the Linux root and swap partitions) before rebooting to go into NT.

Linux and Device Drivers

There are some situations where Linux drivers for the PC hardware being installed are not available. If the drivers are not critical to the Linux boot process, then the installation will complete without any problem. For example, if Linux does not support the Video card or Sound card in the system, this will not affect the installation. However if Linux does not support the IDE controller (for the hard disk), such as an Ultra 66/100 IDE card, then the default RedHat installation will not work. This problem can be solved if there is a kernel patch available for the device in question. Using the patch, a new kernel can be compiled for the boot floppy. Also, new RPMs should be created for the updated kernel and should be added to the Linux distribution. Now the installation can be done on the system in the usual manner. Once you have created a new distribution and boot floppy, you can continue to use this for PCs which may not have the new hardware in them. This means that you only have to maintain one Linux distribution and boot floppy for installing systems. As you add support for new hardware, you are updating the distribution and floppy, which you can use on all systems. Only having to maintain a single boot floppy for many kinds of installations has turned out to be a good thing. In particular it reduces complexity, reduces the work involved in maintaining the system, and reduces confusion.

Alternative Install Techniques

Dual boot installation can be automated using other techniques, in addition to what is described in this paper. Those techniques include

- Cloning the hard disk.
- Copying the NT image during the %post section of kickstart.

In the first method both the operating systems and applications are installed on a "standard PC" in the desired manner. Once the install is done, the disk is kept as a master copy for the install. For any new PC install, a simple disk copy is done onto new PC disk and after host specific customizations installation is complete.

In the second method, kickstart is done pretty much the same way as described in this paper, but during the %post section, instead of installing NT, a previously made NT image is copied to the disk.

Both of these methods provide very good turnaround time for installing PCs in bulk.

However both of these methods have some disadvantages that made us decide to develop our current method. If there are many different hardware configurations, you may need an image per hardware configuration. If there are different types of software configurations required, again, an image per software configuration may be required. If there are many combinations of both software and hardware configurations, then the number of images required can multiply rapidly. If a change is needed, a patch or new software package or whatever, many of these images may require changes. The amount of work required to maintain this seemed too great to be worthwhile. In the case of true installation, one has to upgrade the operating system or application in the distribution only. The next time a new PC is installed, it automatically gets the upgraded OS or application.

The second issue with cloning or copying NT is that all the cloned systems will share the same SID. When these systems are put on the network, Local Administrator on one system will be treated as Administrator on all the cloned systems, because they share same SID, even though they may have different passwords for administrators. This is true for both Administrator and non-Administrator accounts. There is software available which can reset the SID to some random generated value after cloning the disk (like Norton Ghost [3]), but Microsoft does not support resetting the SID.

The advantage of using the method described in this paper is that both operating systems are installed using their native install procedures on the PC where they are going to be used.

Evaluation of the Technique

We have been using this technique to do installs for more than 6 months (as of the writing of this

paper). Our PC install group is very satisfied with this method. We have not faced any major problems although we have found some unsupported hardware in Linux. We solve this problem by recompiling the boot kernel and upgrading the kernel in the distribution. We have been requesting the Linux Kickstart developer community to support a %pre section in the kickstart configuration file, similar to the %post section, with the difference that commands specified in %pre section would be executed before Linux installation begins. In that case we would be able to partition the disk the way we need it from the start, and would not have to jump through hoops to repartition the disk. It appears that a %pre section may be available in a future release of RedHat.

To Do List

Though the system described above works quite nicely, there is room for improvement. Some of the major items in our TODO list are mentioned below:

- Solving the NT hostname problem described earlier.
- Automatically installing/configuring X and the sound card during Linux installation.
- A more ambitious item is to extend the system to install any two (or possibly more) operating systems (especially to support Linux and Windows 2000 dual-boot systems).

Source

The source and complete profiles to do different combinations of operating systems (NT and RedHat Linux) are available from the author by sending email to dualbootinfo@research.bell-labs.com.

Author Information

Rajeev Agrwala is working as a System Administrator for Bell-labs Research for the last four years. He is currently responsible for planning, deploying, and maintaining Unix Systems and process automation. Reach him via US mail at Lucent Technologies; Room 2t-406; 600 Mountain Ave.; Murray hill, NJ 07974. Reach him via electronic mail at rajeeva@lucent.com.

Rob Fulmer has been an Systems Administrator for Bell Labs Research for the last six years. He is currently responsible for maintaining and building the NT infrastructure and for NT process automation. Reach him via US mail at Lucent Technologies; Room 2t-412; 600 Mountain Ave.; Murray hill, NJ 07974. Reach him via electronic mail at rjf@lucent.com.

Shaun Erickson has worked at Lucent Technologies (previously AT&T), for 12 years, the last 5 as a System Administrator for Bell Labs, in Murray Hill, NJ. He can be reached via U.S. Mail at 600-700 Mountain Ave., Room 2C-533, Murray Hill, NJ, 07974-0636. He can be reached electronically at ste@research.bell-labs.com.

References

- [1] Rober Fulmer and Alex Levine (Lucent Technologies, Bell Labs), "Autoinstall for NT: Complete NT Installation over the Network," Usenix, Large Installation System Administration of Windows NT conference, *1998 proceedings*. This paper provides the method to install NT over the network. We use the same technique to install NT in later part of our install.
- [2] RedHat's kickstart user guide available at <http://www.redhat.com/kickstart> explains all the kickstart commands.
- [3] Information on Norton Ghost can be found at <http://www.symantec.com/sabu/ghost/indexB.html>.

Wide Area Network Packet Capture and Analysis

Jon Meek – American Home Products Corporation

ABSTRACT

We describe a system to record and analyze “raw” Frame Relay and point-to-point T-1 packets. The data are captured by “eavesdropping” on the HDLC transmit and receive lines between the router and CSU/DSU. Analysis of the data provides circuit and application utilization information on a one-second or shorter time scale. Routine and custom reports are accessible through Web interfaces to provide easy access by our global systems and network staff. The packet data can also be used to debug applications in the same way as conventional packet capture systems.

Introduction – Why We Needed This System

Frame Relay networks provide organizations with a flexible and economical method of interconnecting sites over a wide range of distances. A major source of the flexibility comes from the ability to connect many circuits over a single access line, such as a T-1 (1.5 Mbps) or E-1 (2Mbps, used in Europe). Each circuit, called a PVC (Permanent Virtual Circuit), has a guaranteed bandwidth, known as CIR (Committed Information Rate). Most Frame Relay carriers allow PVCs to “burst above CIR”, possibly to the full bandwidth of the access line. The sum of the instantaneous bandwidth for all PVCs can not, of course, exceed the bandwidth of the access line. This leads to interesting traffic management questions.

Complex Frame Relay networks are often laid out in a “hub and spoke” arrangement. Multiple hubs may connect subsidiary offices in a geographical area. The hubs are then joined together, usually with higher bandwidth interconnections.

While debugging Frame Relay network problems, for both bandwidth management and application issues, we have used tcpdump [McCa97] to record packets at the Ethernet interface of routers. We often wished, however, that we could see exactly what data were flowing in and out of the T-1/E-1 serial access lines. This was especially true at Frame Relay hub sites where many packets pass through the router, but never appear on the Ethernet side because they are destined for another site on our network. In addition, useful Frame Relay header information is lost once the frames are converted to Ethernet packets.

As we did more application debugging and traffic analysis it became clear that we needed a system to record raw frames outside the router, directly from the communications lines. Then we could examine any of the Frame Relay header information and as much of the data, including IP header and payload, as we cared to record.

Commercial systems were reviewed but none were found that met the requirement to record raw

Frame Relay packets for more than a few minutes. Our company already used two of the more popular brands of “WAN Probes”, but they are mostly useful for real-time diagnostics, and RMON (Remote Network Monitoring Management Information Base) type historical data. We considered using Network Flight Recorder [Ranu97], but at the time, it could not record data from WAN communications lines.

While most routers count the Frame Relay congestion notification bits (FECN and BECN, Forward and Backward Explicit Congestion Notification) in the header, they do not count discard eligible (DE) bits. The five-minute counts of FECNs and BECNs that we record via SNMP do not provide any method to assign the occurrence to a particular second, or to particular packets. Debugging an application / network interaction problem without the raw packet data is very difficult.

In this paper we will first review the hardware requirements and packet acquisition software. Then the traffic analysis software will be discussed, followed by real-world analysis examples including “Congestion and Circuit Capacity Planning”, “Using the Raw Packet Data”, and “Application Profiling”. Two short sections describe the extension of the system for T-1 point-to-point circuits, and using tcpdump to perform similar analysis when the packets of interest are available on a LAN. We will close with some ideas for future applications.

The Hardware

The system is built on a low cost desktop platform running RedHat Linux (version 5.2 or 6.x). The heart of the hardware is one or more communications boards from Sangoma Technologies Inc. (Markham, Ontario, Canada). The first few monitors we built used two Sangoma WANPIPE S508 ISA cards, but we are now using a single Sangoma WANPIPE S5142 PCI card that can handle four communications lines at up to 4Mbps in “listen-only” mode.

Acquiring the bi-directional data requires the use of two receive lines and the associated clock signals

on the Sangoma cards. The transmit lines of the card are not connected. While we have successfully connected to T-1/E-1 lines using short cables directly attached to the communications lines, the use of an active "Multi-Interface Tap" is recommended. These taps present a high-impedance to the signal lines and allow a long cable to be safely used between the tap and the computer. The cost for a system to monitor a single T-1/E-1, including PC, communications board, and WAN tap is about US\$2000. A second T-1/E-1 can be monitored on the same PC for an additional US\$800.

Acquisition and Analysis

The basic model for the system is to record packets in both directions (in-bound and out-bound) for fifteen-minute periods. At the end of each period the packet files are closed and a new pair of acquisition processes are started. Then a summary program processes the data from the previous period.

This model provides considerable simplification at the cost of only about a fifteen-minute delay compared to a real time system. It also allows convenient packaging of the summary results and a method to locate raw packet data when deeper analysis is needed. The hardware requirements are lower for this post-process model; all that is necessary is for the summarization process to complete in less than fifteen minutes while handling the input streams without packet loss.

Acquisition Software

The software consists of drivers provided by Sangoma, a modified version of Sangoma's example C program for data capture, and a set of Perl programs to control the acquisition and analyze the acquired data.

Sangoma's driver software was set up for CHDLC (Cisco HDLC) mode. The packet capture program, `frpcap`, writes files in a format closely modeled after the `tcpdump` format. The only significant difference is that the Frame Relay header is recorded in place of the Ethernet header. The first 150 bytes of each raw packet are usually saved to provide context information during application analysis.

The packet acquisition process is driven by `frcap_run`, a Perl script that runs every fifteen minutes. `frcap_run` stops the current `frpcap` processes (one for each data direction) and immediately starts another pair. A traffic summary program, `fr_decode`, is then run on the two packet files. Following the summarization the raw packet files are compressed, and any files that are older than a preset age are deleted to conserve disk space. For a fairly busy set of Frame Relay circuits on a T-1 access line, eight days worth of raw packet files consumes 3-6GB of disk space.

Analysis Software

The fifteen-minute raw packet files are summarized by a Perl program that appends its output to a daily summary file in XML format. The XML file is read by other programs for display and further analysis. An example of a fifteen-minute summary output displayed by a Web application that formats the XML data is shown in Figure 1. To reduce the size of Figure 1, data for a minor PVC were removed and only the top five numbers, rather than our usual ten, are shown in each category.

The report consists of five major sections. The first section is a PVC summary showing the DLCI (circuit number), number of packets and bytes, percentage of bytes per circuit relative to all circuits on the access line, congestion notification counts, DE (discard eligible) counts and TCP re-transmission counts. Since this router does not set any congestion notification information (Frame Relay switches further down stream set these) or DE bits, the counts are all zero for the out-bound direction. Some of our routers do set DE for Internet traffic to give it a lower priority (setting DE tells the carrier that the traffic can be dropped if the network is congested, in exchange the carrier does not count the packets towards certain credit limits). The TCP re-transmit counts are done separately for packets with and without DE so that we can determine the effect on packet loss within the Frame Relay network when DE is set.

Layer 2 and 3 protocol counts are summarized in the second section. For each protocol observed the Ethernet/802.2 type field, IP type by number (if an IP protocol), protocol name, number of packets and bytes, and percent utilization by bytes for the PVC are shown. For TCP/IP a count of packet re-transmissions is displayed.

In the third section we display the busiest and quietest seconds for the access line and for each PVC. The generation and use of these data are described later under Congestion and Circuit Capacity Planning.

The top sources and destinations of data are shown in the fourth section. The protocol, IP address, port number, number of bytes, and percentage of the total for all traffic on the access line are included. If the server name is known it is also displayed, along with a short description of the application.

The fifth section of the report lists the utilization of the access line by application. Where possible we identify applications by the IP address of the server. Although it might make sense to further define applications by port number, many of our servers run a single application. In fact we often have two servers running the same application in which case packets with either IP address will count towards the application. Database servers often run multiple applications which all use the same IP address/port number pair so the addition of a port number qualification would still not uniquely identify all applications. In the future we

Frame Relay Traffic Summary (Philadelphia)

Out-Bound from Philadelphia Data

Capture Time: Thu Feb 10 11:00:00 2000 - Thu Feb 10 11:15:01 2000 GMT

PVC Summary (Out-Bound from Philadelphia)

DLCI	Packets	Bytes	%	FECNs	BECNs	DEs	DE	No DE	
460	79,648	12,422,725	30.2 %	0 0.0 %	0 0.0 %	0	0	328	London
490	119,404	28,677,448	69.8 %	0 0.0 %	0 0.0 %	0	0	1,321	Paris
All	199,052	41,100,173							

Protocol Counts (Out-Bound from Philadelphia)

DLCI	Protocol	Packets	Bytes	% of PVC	TCP ReTransmits
460	London				
	0800 06 IP TCP	40,291	9,068,685	(73.0%)	328 (0.8%)
	8137 IPX	34,675	2,805,741	(22.6%)	
	0800 11 IP UDP	1,671	316,118	(2.5%)	
	0800 01 IP ICMP	2,593	197,600	(1.6%)	
	809b ATALK	203	15,000	(0.1%)	
	0800 58 IP IGRP	200	14,616	(0.1%)	
490	Paris				
	0800 06 IP TCP	70,203	21,871,361	(76.3%)	1321 (1.9%)
	8137 IPX	46,048	6,228,881	(21.7%)	
	0800 11 IP UDP	2,051	498,644	(1.7%)	
	0800 01 IP ICMP	882	58,936	(0.2%)	
	0800 58 IP IGRP	205	14,886	(0.1%)	

Access Line Busiest Seconds (Out-Bound from Philadelphia)

Time	Bytes	kbps
11:08:11	125,513	1,004.1
11:08:09	118,855	950.8
11:08:13	116,336	930.7
11:02:53	108,926	871.4
11:08:14	104,754	838.0

PVC Busiest Seconds (Out-Bound from Philadelphia)

460	London		
	11:02:53	77,873	623.0
	11:02:52	76,221	609.8
	11:02:54	47,667	381.3
	11:02:56	46,748	374.0
	11:00:07	44,487	355.9
490	Paris		
	11:08:11	112,854	902.8
	11:08:13	105,761	846.1
	11:08:09	95,425	763.4
	11:08:14	92,765	742.1
	11:08:10	85,951	687.6

Access Line Quiet Seconds (Out-Bound from Philadelphia)

11:12:53	11,366	90.9
11:12:52	14,118	112.9
11:12:54	15,371	123.0
11:12:55	22,993	183.9
11:11:48	23,544	188.4

PVC Quiet Seconds (Out-Bound from Philadelphia)

460	London		
	11:05:14	3,640	29.1
	11:04:55	3,859	30.9
	11:05:33	4,068	32.5
	11:07:48	4,118	32.9
	11:06:20	4,170	33.4

Figure 1a: Frame relay traffic summary for a single T-1 access line

```

490 Paris
    11:12:53      3,460      27.7
    11:12:54      6,613      52.9
    11:12:52      8,187      65.5
    11:13:18     14,021     112.2
    11:12:55     14,065     112.5

Top Sources (Out-Bound from Philadelphia)
      Bytes % of Total
1 TCP 155.94.114.164 1867 4,673,580 11.0 Philadelphia GroupWise
2 TCP 10.2.71.201 1494 2,644,817 6.2
3 TCP 155.94.155.23 1521 1,671,696 3.9 ra01u04 - Philadelphia DCG
4 TCP 192.233.80.5 80 1,272,224 3.0
5 TCP 209.58.93.100 1494 931,341 2.2 MARTE WinFrame 1

Top Destinations (Out-Bound from Philadelphia)
      Bytes % of Total
1 TCP 10.248.107.217 7100 4,742,966 11.2
2 TCP 10.247.113.201 4498 1,272,224 3.0
3 IPX 0451 01000105 1 1,138,074 2.7 NCP
4 TCP 10.248.89.1 7100 952,921 2.2
5 TCP 10.247.66.76 1073 931,341 2.2

Application Summary (All PVCs, Out-Bound from Philadelphia)
      New TCP      Total TCP
Application      Sessions      Sessions      Packets      Bytes

Internet TCP      4,684      4,817      41,455      13,128,752
IPX                1,016      1,167      90,631      9,785,697
Unknown TCP        98         138      41,305      7,141,942
GroupWise TCP      138        150      12,106      6,722,084
DCG TCP            2          5       7,370      1,824,223
MARTE WinFrame TCP 2           5       4,428      1,041,713
IP Protocol 0b NVP-II 13         20      3,894      839,902
EDMS TCP           13         20      3,075      775,923
MARTE Oracle TCP   1          3       1,882      472,541
MLIMS TCP          38         22       850       255,473
Internet ICMP      3,255      214,690
Unknown ICMP       780       78,666
ProbMan TCP        0          4       181       48,826
IP Protocol 3a IPv6-ICMP 598      43,012
Unknown ATALK      203       15,000
ASTROS TCP         2          6        62       9,963

TCP SYNs (connection requests): 5996      Total TCP Re-Transmissions: 1662

```

Figure 1b: Frame relay traffic summary for a single T-1 access line (cont.)

hope to encourage the use of virtual IP addresses assigned on a per application basis to provide a simple accounting method. The "New TCP Sessions" column indicates how many sessions were initiated during the fifteen-minute period and "Total TCP Sessions" counts both new and ongoing sessions. For Web based applications these counts are not very useful except as a type of hit counter, but for applications with persistent connections it is a measure of the number of users connected during the period.

The final two items in the report are a count of initial TCP SYNs, which might be used for intrusion detection, and a count of TCP re-transmissions for all PVCs.

Figure 2 shows the top of a report for the in bound direction. Note that we have a variety of FECN, BECN, and DE information related to packets flowing in this direction. The FECNs and BECNs provide

information on the level of congestion in the carrier's Frame Relay network [Blac95]. The rest of the report contains the same information as shown in Figure 1.

Several customizable reports using the fifteen-minute summary data are available. A Web form can be used to select a single application and then generate a list of "Application Summary" entries for just that application. This list of application usage is very helpful for determining the bandwidth impact of an application and the number of users. Since we are often told "There will be 400 users in Europe accessing this Philadelphia based application", we can use the tool to judge how many are logged-in simultaneously and to monitor usage growth. Another program will generate a daily or monthly "Application Summary" by summing usage for each application over time.

The above example data is for one of our busier access lines, but it is far from the most complex. One

European Frame Relay hub site has 13 PVCs on a single access line. Some of the traffic is business critical telnet traffic between subsidiary sites and an AS/400 at the hub site. Much of the traffic, however, is intranet mail or Internet data that comes in on one PVC and then heads towards the US on another PVC. Without the Frame Relay monitor it would be very difficult to determine what applications and protocols consume the PVC and access line bandwidth. In some cases it is necessary to obtain application summaries for a single PVC to determine what is happening on a particular circuit. We do not routinely report application and protocol usage on a per PVC basis in order to keep the size and complexity of the reports reasonable.

Congestion and Circuit Capacity Planning

Like many organizations, we collect router statistics via SNMP every five minutes. While five-minute averages are a useful measure of how a circuit is doing in relationship to its bandwidth limit, they do not tell a lot about the instantaneous (one second, or smaller, time scale) state of a circuit that governs interactive performance. If a circuit is saturated for several ten-second bursts during a single five-minute period the average utilization might appear to be quite reasonable. An interactive user, however, would likely say that the network was slow while his packets were waiting in a router buffer.

Our monitor attempts to measure the largest peaks in a fifteen-minute period by summing the number of bytes transmitted and received for each one-second interval. The summary program reports the ten busiest seconds for the access line and each circuit. A sample plot of busy seconds for one day is shown in Figure 3. The presence of many long vertical lines connecting the data points is good because it indicates a large variation in the top ten busy seconds and therefore there are fewer than ten very congested seconds in a fifteen-minute period. The lack of long vertical lines around 13:30 GMT indicates significant ongoing congestion. Quiet seconds are measured in a similar fashion. If a usually busy circuit has seconds with zero, or a low number of bytes, then it may indicate a circuit or routing problem.

We had one problem where a large number of quiet seconds were showing up with low utilization during busy times of the day. Since there were also user complaints, a detailed analysis of the packet data

was performed. It showed all IP traffic periodically stopping for about eight seconds while IPX was still flowing normally. We traced the root cause to an IP routing problem. Without the IPX traffic the problem would have been easy to spot since there would have been periods of about eight seconds with zero traffic on an otherwise busy circuit.

While we have not yet developed a formal rule set, it should be possible to determine how well PVCs and access lines are sized with respect to the actual traffic using busy second data. Clearly, looking at the busiest seconds on a circuit is much more meaningful than five-minute average data when mission critical interactive applications are the most important traffic.

Using the Raw Packet Data

Since the raw packet data are stored in separate files for in-bound and out-bound directions the two files must be combined for traditional packet trace analysis. A utility program performs this task by putting the packets from a pair of files into a time ordered sequence and writing a tcpdump format file. A "Frame Relay information" file containing the Frame Relay header information is also written.

We have a packet trace analysis program originally written for tcpdump files that can optionally read the "Frame Relay information" file and list the DLCI (circuit ID), FECN, BECN, and DE bits for each packet. Using this feature, we have discovered that some applications were operating over asymmetric routes. In addition to our own analysis programs, the tcpdump format files can be examined using other programs such as tcpdump itself, or ethereal [Ether00]. Since ethereal, and its companion program editcap, can export packet data to other formats, the traces can be analyzed with popular commercial products. When a session needs to be followed through multiple fifteen-minute periods we use a simple program to concatenate multiple tcpdump files.

In a previous paper [Meek98] we discussed interesting issues we have had with our telecommunications vendors. Our complaints about a slow circuit sometimes yield a vendor response like: "Customer is exceeding CIR by 160%" with the implication that the over utilization of a circuit (bursting) has lasted for an excessively long period. With raw packet information it should be possible to compute bandwidth utilization

In-Bound to Philadelphia Data													
Capture Time: Thu Feb 10 11:00:00 2000 - Thu Feb 10 11:15:01 2000 GMT													
PVC Summary (In-Bound to Philadelphia)													
DLCI	Packets	Bytes	%	FECNs		BECNs		DEs	DE	No	DE		
460	62,915	6,453,502	36.9 %	515	0.8 %	31	0.0 %	2,211	6	111	London		
490	109,900	11,024,584	63.1 %	39	0.0 %	11,800	10.7 %	0	0	656	Paris		
All	172,815	17,478,086											

Figure 2: Portion of in-bound frame relay traffic summary for a single T-1 access line.

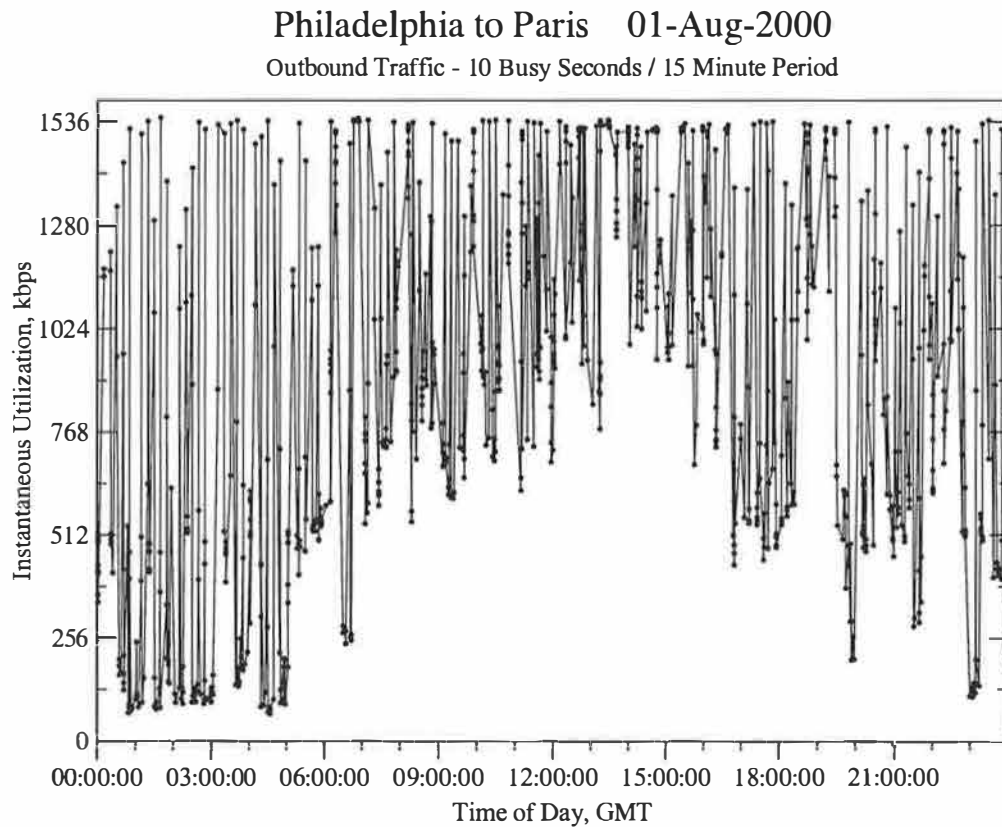


Figure 3: Busy seconds on a frame relay circuit. This circuit has a CIR of 1024 kbps on a T-1 access line.

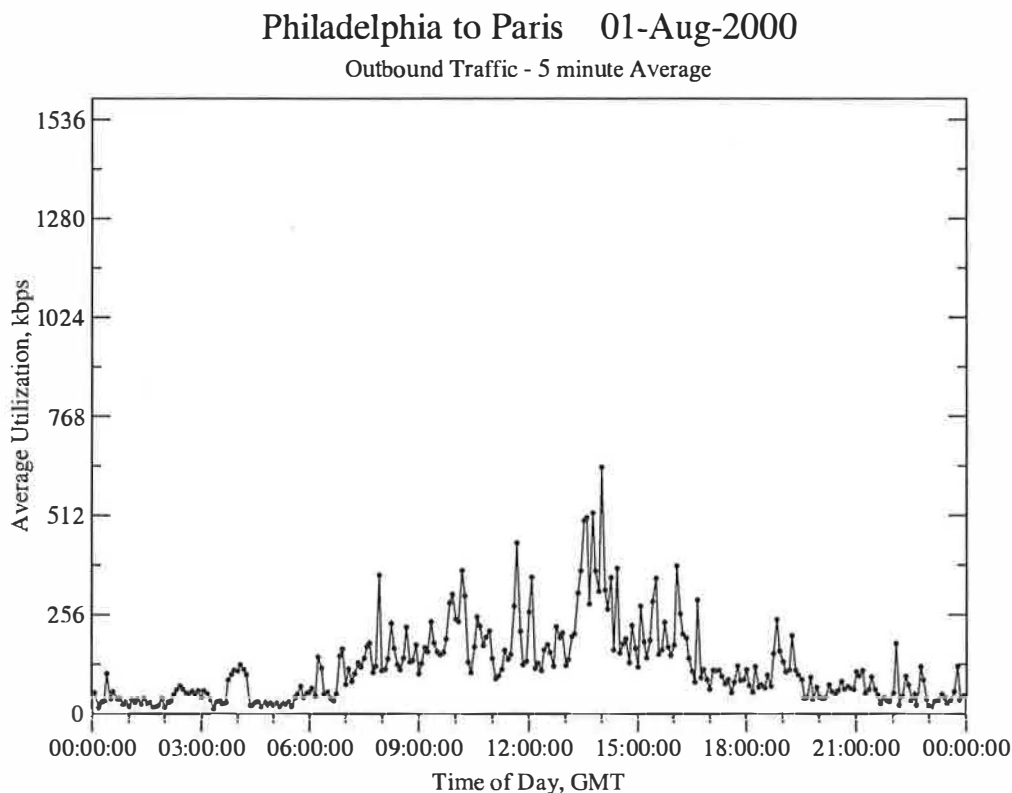


Figure 4: Five-minute average traffic for the same day and circuit as Figure 3. This standard method of measuring utilization hides many significant traffic spikes that are observed in the busy seconds plot.

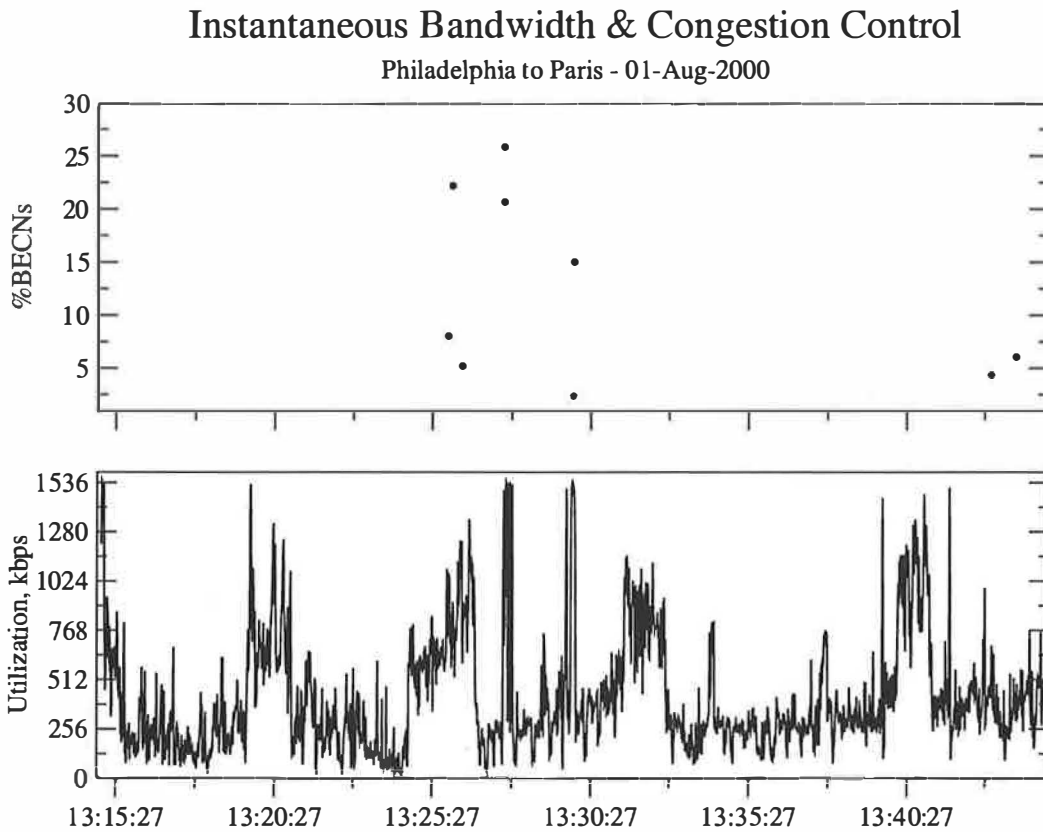


Figure 5: Instantaneous bandwidth utilization (one-second time scale) and Frame Relay network congestion control information (percent of incoming packets with BECN set).

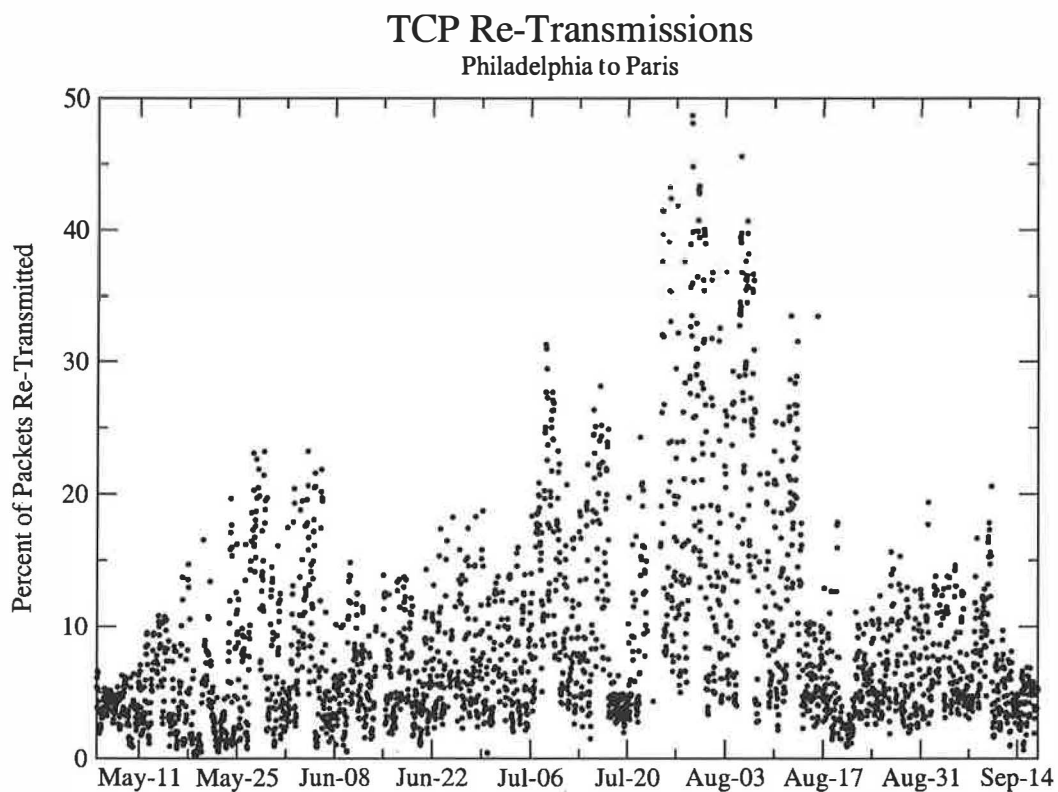


Figure 6: Percentage of TCP packets requiring re-transmission per hour.

on a per-second basis and use the same algorithm as the telecommunications equipment to verify when, and by how much, CIR was exceeded. Figure 5. shows the per-second bandwidth utilization of a circuit for a thirty-minute period and the percentage of in-bound packets with the BECN bit set to indicate congestion on the out-bound circuit. The BECNs are sent by the carrier's switches to indicate that there is congestion in our out-bound direction and that our bandwidth usage will be throttled if we are exceeding CIR and are out of credits. In the future we hope to use this data to accurately determine when we truly exceed our contracted bandwidth and how we might implement quality of service to prioritize traffic to manage bandwidth bursts.

Packet loss is an important parameter in any data network. One measure of packet loss is the percentage of TCP packets that must be re-transmitted. Figure 6. illustrates TCP re-transmission rates on one of our busy circuits over several months. Hours with fewer than 10,000 TCP packets are not shown. Since this circuit feeds multiple downstream Frame Relay circuits, and many LAN segments, packet loss could occur in several places. During late July we had problems with a T-1 interface that caused a significant portion of the re-transmissions during that period. Further analysis of the raw packet data can determine what destination IP addresses were responsible for the re-transmissions. We recently added a new section to the standard report (Figure 1) that lists the top ten destinations of re-transmitted packets to help identify hosts or subnets with problems. Re-transmitted packets are counted by

tracking the TCP sequence numbers by session for packets with a payload (not acknowledgement-only packets).

Application Profiling

Profiling the bandwidth requirements of an application is a useful exercise to perform during the development or evaluation of a software product. We have often found that applications originally developed for use on a LAN have serious problems when they are moved to a WAN or Internet environment. Problems result from large quantities of data being sent, application level handshaking resulting in excessive round-trip-time waiting, or even the same set of data being requested (and delivered) multiple times due to a programming error. These issues have occurred in both internally developed and purchased commercial software.

One advantage to using the WAN packet capture system for application profiling is that the application under test can be observed along with all of the other data flowing on the circuit at the same time. Since we routinely capture all of the traffic on monitored circuits, no advance preparation is required for most application profiling tests. We have found, however, that it is helpful when testing an interactive application if the test procedure has timed pauses of 15 to 30 seconds where the user does not touch their hardware. The pauses are used to separate phases of the application in the packet traces and to determine if the client and server "chatter" when idle.

AS/400 "Green Screen" Session

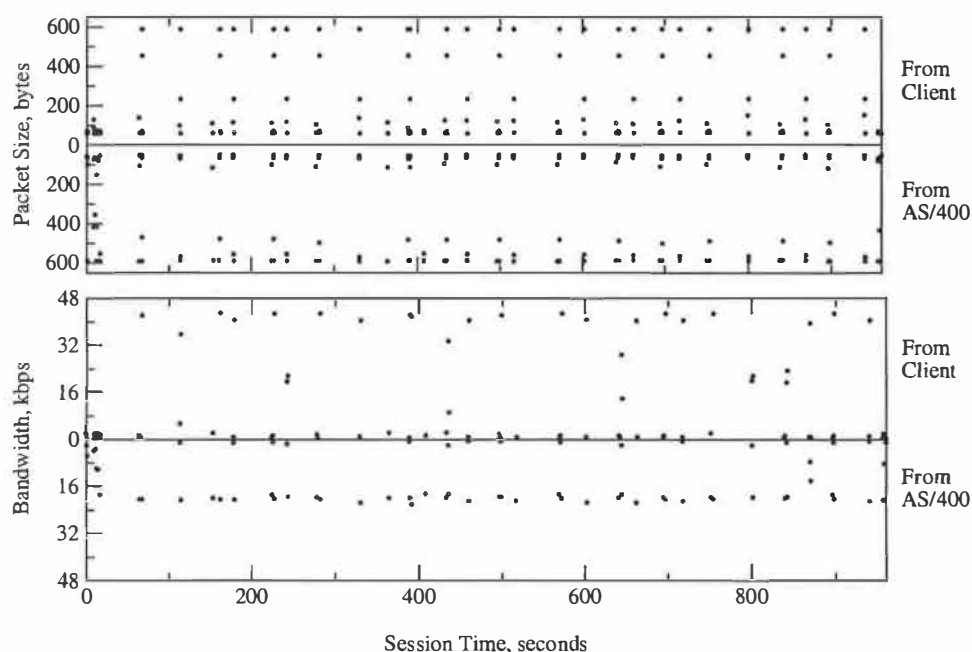


Figure 7: IP packet data for a single session extracted from a series of Frame Relay packet capture files.

A disadvantage to using the WAN packet capture system for these tests is that we generally capture only 150 bytes of combined header and payload. While this small number of payload bytes is often enough to determine the context of the application (especially when ASCII or EBCDIC data are involved), it might not be enough to confidently determine that multiple packets with identical payload are being transmitted within a session (determined by computing MD5 checksums on payload content [Meek98]).

An example of application profiling is summarized graphically in Figure 7. The application is a widely used ERP (Enterprise Resource Management) system that uses page-based terminals as the user interface. The user interacts with the system by filling out text forms and then transmitting the screen page to the server. This method, similar to modern Web applications, is an efficient way to implement interactive applications on a WAN since reasonable size chunks of data, in comparison to keystrokes, are transmitted at one time.

To determine how much bandwidth a single user consumes we look at the packets for individual sessions as a function of time. The top plot shows the distribution of packet sizes for each direction, while the bottom plot shows the bandwidth used per second. A detailed look at the numbers shows that 143,473 bytes in 400 packets were sent from client to server and 112,560 bytes in 348 packets were sent from server to the client. By closely inspecting the data using an interactive data analysis tool [Grace00] we find that a typical transaction is completed in one or two seconds. The maximum bandwidth used was 43kbps from client to server and 22kbps in the other direction.

The packet data in this analysis were extracted from 30 minutes of raw packet capture data from a

Frame Relay access circuit with 13 PVCs. The first step was to combine in-bound and out-bound packet streams into single tcpdump format files. The tcpdump files for the two 15-minute time periods required to span the session were then concatenated. Finally, the combined file was processed with tcpdump acting as a pre-filter to select the session of interest and feed it to our own software that summarized the session and prepared the packet and bandwidth data for plotting.

While the tools presented here do a complete job of quantifying the actual bandwidth used by an application on a one-second time scale, but they do not address simulated scaling of the application. Presumably, our data could be used as input to a network simulation tool to perform the scaling simulation. Our tools can, however, select any slice of actual recorded network traffic based on source, destination, port number, etc. and determine the total bandwidth utilization for the applications encompassed by the slice.

In order to determine how the use of an application changes over time we can look at some parameter representing usage. In Figure 8 we show the number of sessions per week for one application. The use of this particular application varies depending on business cycles and holiday schedules. Other parameters useful as a measure of application usage are bytes transferred or number of packets. Bytes or packets are especially useful for Web, or other non-session oriented applications.

Extension to T-1 Point-to-Point Circuits

We were pleasantly surprised to discover that the hardware and software can be used without modification on T-1 point-to-point circuits. The packets on the point-to-point circuits have a header very similar to a Frame Relay header. The first two bytes of the Frame

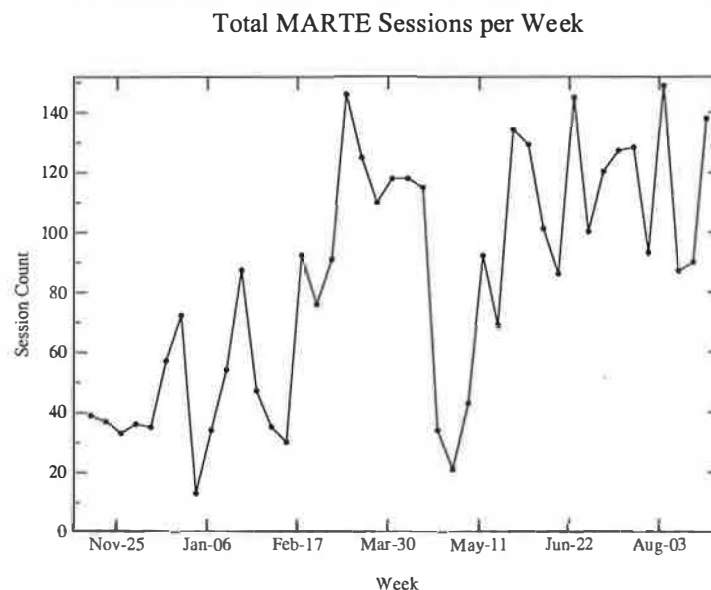


Figure 8: Usage pattern of a single application by measuring the number of sessions per week.

Relay header contain packed data representing the DLCI number and the FECN, BECN, and DE bits. [Blac95] The first byte of a point-to-point serial line packet is set to 0x0F for unicast packets and 0x8F for "broadcast packets" and the second byte is always zero. [Cisc00] For both Frame Relay and point-to-point serial the third byte contains the Ethernet protocol code. Note that these conventions may be specific to certain types of equipment.

Related Techniques

When all traffic of interest is accessible from the LAN, simpler tools and techniques should be used to record traffic. We use a simple script to start tcpdump and rotate the packet capture files on a time schedule controlled by cron (the user-mode command scheduler). The acquired data can be analyzed using the methods described here for WAN traffic.

Future Work

Because we are now moving some of our major circuits to ATM in order to overcome T-1/E-1 and Frame Relay bandwidth limitations we hope to be able to extend the techniques discussed here to ATM. This should be straightforward if the ATM interface card performs the re-assembly of ATM cells into complete IP packets.

Some of the parameters measured by the system will probably be used to generate alarms when they exceed certain thresholds. The TCP re-transmission rate and quiet/busy seconds are likely candidates for alarms. Frame Relay provides information about circuits using LMI (Local Management Interface) packets. Currently we do not decode these, but plan to add the capability in the future.

We would like to measure the effectiveness of QoS (Quality-of-Service) schemes by measuring packet delays through the router for packets in different classes of service. This would likely be done using tcpdump on the Ethernet side and WAN packet capture on the serial line side of the router. Histograms of packet delays during busy periods should show that the high-priority traffic passes through the router more quickly than lower priority traffic. The results might be used to tune QoS parameters.

Conclusion

We have assembled a low cost Frame Relay and T-1 packet capture system with a large memory and applied it to real problems. The use of this monitor is helping us communicate to management how the WAN is being used at the application level. It also provides detailed worst-case traffic information through the analysis of busy and quiet seconds. We have upgraded bandwidth on some circuits following analysis of peak utilization data, identified unusual routing problems, and profiled the network impact of applications. The large memory and long retention time for the raw packet data allow us to troubleshoot many

network problems days after they occurred, an important factor in our large, global organization. The data analysis discussed in this paper just touches on possible uses of raw packet data from WAN circuits. In the future, we expect to mine the information in new ways.

Availability

Supplemental information and some of the software used in the work described here can be obtained at <http://wanpcap.sourceforge.net>.

Acknowledgments

The author would like to acknowledge Jim Trocki, Kevin Carroll, Kim Takayama, Jim French, and the technical staff at Sangoma Technologies Inc. for valuable discussions, advice, and information during the development of the tools. Drafts of this paper were expertly reviewed by Bill Brooks, Kevin Carroll, Edwin Eichert, and William LeFebvre.

Author Information

Jon Meek is Senior Consultant in the Border Network Services Group at American Home Products Corporation. He received BS and MS Degrees in Physics, and a PhD in Chemical Physics all from Indiana University and has worked in Experimental Nuclear and Chemical Physics, Analytical Chemistry, and Information Technology. His recent research interests include applying Web technology to scientific and network management applications, systems and network management, data integrity, and data acquisition. He can be reached at [<meekj@pt.ahp.com>](mailto:meekj@pt.ahp.com) or [<meekj@ieee.org>](mailto:meekj@ieee.org).

References

- [Blac95] Uyless Black, *Frame Relay Networks: Specifications and Implementations*, McGraw-Hill 1995.
- [Cisc00] Cisco Systems, WAN Group, private communication.
- [Ether00] Ethereal, A network protocol analyzer, <http://ethereal.zing.org/>, 2000.
- [Grace00] "Grace, a WYSIWYG 2D plotting tool for the X Window System and M*tif", <http://plasma-gate.weizmann.ac.il/Grace/>, 2000.
- [McCa97] Steve McCanne, Craig Leres, Van Jacobson, "TCPDUMP 3.4", Lawrence Berkeley National Laboratory Network Research Group, 1997.
- [Meek98] Jon T. Meek, Edwin S. Eichert, Kim Takayama, "Wide Area Network Ecology," *Proceedings of the Twelfth Systems Administration Conference (LISA '98)*, USENIX, Boston, 1998
- [Ranu97] Marcus J. Ranum, Kent Landfield, Mike Stolarchuk, Mark Sienkiewicz, Andrew Lambeth, and Eric Wall. "Implementing a Generalized Tool for Network Monitoring," *11th Systems Administration Conference (LISA)*, 1997.

Sequencing of Configuration Operations for IP Networks

P. Krishnan – ISPsoft, Inc.
Tejas Naik – Bell Laboratories¹
Ganesan Ramu – CoSine Comm., Inc.
Roshan Sequeira – ISPsoft, Inc.

ABSTRACT

When configuring an IP network, changes may need to be made to several devices as part of one configuration operation. Given an in-band access to the devices where data and control packets flow on the same network, it becomes imperative to sequence the changes intelligently to retain connectivity to the devices that need to be reconfigured. In this paper, we present techniques to sequence configuration operations for IP networks. We describe our experiences with implementation of the techniques, and experimentally evaluate the usefulness of the proposed techniques. We demonstrate that automatic sequencing is valuable in IP network configuration.

Introduction

A basic operation performed during network operations and management is the configuration of the network. Many configuration and reconfiguration operations are typically performed from a remote management station. To maintain consistency, several devices may need to be updated as part of one configuration operation. To complete the configuration action, the devices to be updated must continue to be reachable from the management station. Reconfiguring some devices may jeopardize the connectivity to other devices that need to be configured. *Sequencing* is the process of determining the order of updates to ensure a successful configuration.

In this paper, we present one of the first studies of automated sequencing in the context of router² reconfigurations in IP networks. Many deployments today use an in-band access to the routers for configuration, where the control and data packets flow on the same network³. In this case, configuration of a router is done through a telnet to the router. Control (configuration information) and data go over the same physical and logical network, directed by the same routing tables. Parameters that an administrator can reconfigure include what we call *critical* and *non-critical* parameters. Changing critical parameters affects how the router computes and exchanges routing information with other routers, and consequently threatens continued connectivity to the router and other routers. Examples of critical parameters include routing protocol parameters, like the Open Shortest Path First

(OSPF) [10, 12] *hello interval*, *router dead interval*, *authentication*, and *authentication key* parameters, fundamental parameters like IP addresses and subnet masks, some access list configurations, etc. Non-critical parameters, on the other hand, do not affect connectivity. Examples include parameters like the OSPF *maximum paths* and *cost* parameters.

One can see that routers can be updated in any arbitrary order while changing non-critical parameters, or when an out-of-band access is available to all the routers. In this case, by out-of-band we mean access through the console or modem ports of the router. However, the order in which the updates are committed (i.e., sequencing) becomes important. Administrators currently try to deduce a sequence manually from their knowledge of the network topology and their understanding of protocols. No automated techniques have been published for sequencing.

In this paper, we study the problem of sequencing when updating critical parameters, and present automated methods to compute an order in which to update the routers. We have implemented the techniques in Java, and conducted experiments that quantify the usefulness of our techniques and the convenience they can provide to the administrator. We demonstrate that our automated sequencing techniques speed up network configuration and make it more reliable.

The rest of the paper is organized as follows. The sequencing problem is explored in more detail in the next section. Our algorithms for sequencing router configurations are presented thereafter. The implementation issues we encountered are presented in the implementation section which is followed by a section on experiments and results. We then conclude.

Sequencing: The Problem and Assumptions

To illustrate the problem of sequencing more concretely, consider the network in Figure 1. Let us

¹Portions of this work were performed while all authors were at Bell Labs.

²We use the term *router* to refer to any remotely configurable network element.

³A few routers in the network may have an out-of-band connectivity through the modem or console port of the router. Some establishments provide an out-of-band access to all routers in their network.

assume that all depicted routers are running the OSPF routing protocol, and are in the same OSPF Area. (See Appendix 2 for a brief description of OSPF.) Let us assume that we want to change the OSPF authentication parameter. The OSPF authentication parameter should be set consistently for all routers in an OSPF area to ensure that they exchange routing information. If during configuration, we update router R before all routers in the set S , there is a possibility that routers in set S may become unreachable from the network operations center (NOC)⁴. Sequencing ensures that the routers that remain to be updated will continue to be reachable. In our example, updating all routers in set S before router R is a viable strategy.

In our study of sequencing during IP network configuration, we make certain assumptions. We assume that the changes are being made to the routers from a remote centralized network operations center and that all routers are initially reachable. Further, the changes to be made are correct; i.e., if the desired changes are committed to the routers, the routing will stabilize soon enough, based on the routing protocol behavior, and to a state as desired by the administrator. We refer to this as the *correctness assumption*. The reachability of routers can be affected by several factors, some of which are outside the scope of any software tool (e.g., power failure at a router). Such issues are not considered as part of this study of sequencing. We assume that any unreachability during the configuration operation is only caused due to the configuration changes being made by the administrator.

⁴We use the terms *network operations center* and *management station* interchangeably in the paper.

Situations may arise where no sequencing is possible. Remote configuration of a set of routers could be theoretically impossible, under some conditions, mostly stemming from asymmetric routing. Consider, for example, the situation in Figure 2.

Assume that the routes are set up (e.g., via static routes) so that all packets can only move clockwise around the circle C of routers. If a critical update is made to any router in the circle of routers, one of the paths (either to the router from the management station, or from the router to the management station) is broken, disabling the ability to reach and configure the other routers in the circle. Such a routing topology is unlikely in practice, since most people use routing protocols that will adapt to network change. However, this gives some insight into the theoretical limits of remote in-band configuration and sequencing in general. For simplicity, we assume in the following discussion that the routing in the network is symmetric (although such a restriction is not really needed for all our techniques), and elaborate more on such situations later in a section on asymmetry in routing.

An alternative approach to the problem of sequencing is to have built-in support in all routers in the network for scheduling configuration changes. This means that the software running on the routers (e.g., the Cisco IOS) must support the scheduling feature. In this scenario, the required configuration changes are uploaded to the router and a timer is set which determines when the changes are executed. Ongoing work in the area of scheduling and scripting management information bases [8, 9] can aid in this problem. However, routers in the marketplace do not

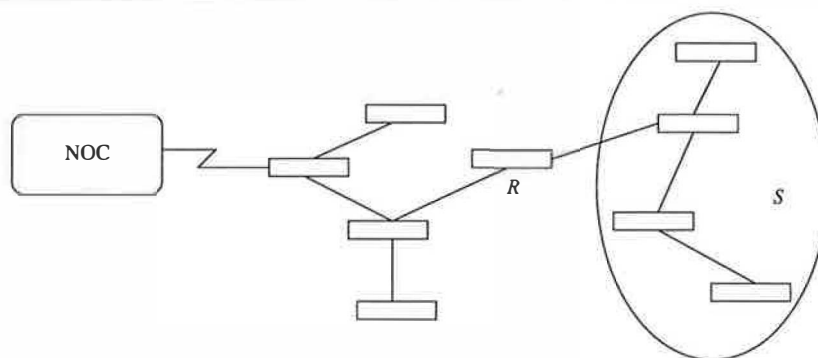


Figure 1: A network of routers and its network operations center (NOC).

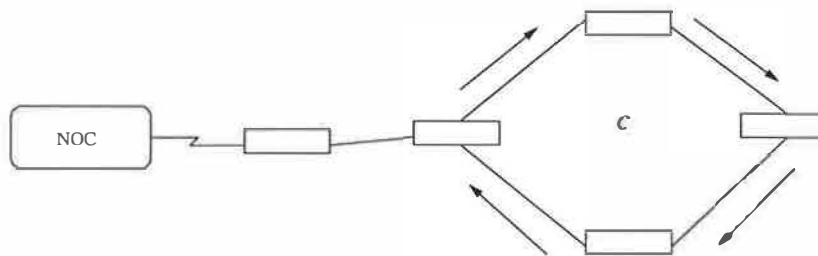


Figure 2: Circular dependencies and sequencing.

provide such a feature. Given that in-band remote configuration is becoming increasingly popular for IP networks, developing smart automated sequencing techniques is essential.

Context for our Study

This study of sequencing arose in the context of an IP network configuration tool [5, 13] that was built at Bell Labs. The tool helps in configuring IP routers, treating them as a part of the network, rather than considering them as stand-alone devices. The tool ensures consistency of configuration and updates several routers as part of a configuration operation, if necessary. The protocol used to contact the routers for configuration is orthogonal to the issue of sequencing. Although there are several proposed standards for management that can be used for configuration [3, 4], most deployed routers today provide a command-line interface for configuration. Configuration commands typically take effect as soon as they are executed.

Our network configuration tool provides a GUI to the administrator for configuration, and hides all its interactions with the network and the routers from the administrator. To interact with the routers for configuration, the tool stores router passwords in an encrypted form in its database, and uses these passwords to login to the routers.

Before we present our sequencing techniques, we introduce the notion of access to a router via *indirect telnet*.

Access via Indirect Telnet

An indirect telnet is essentially a proxy telnet. The basic idea in indirect telnet is to access a (possibly unreachable) router R through an intermediate router N , by first telnet'ing to the intermediate router N and then initiating a telnet to router R from router N .

The usefulness of indirect telnet is best illustrated by the case where routers R and N have a directly connected interface. Since directly connected IP-configured interfaces are visible to each other, if

router N is reachable, then we can access router R through its neighbor N . Such a technique for reaching routers is sometimes used by administrators while configuring routers manually. The concept can be extended to hop through several intermediate nodes en-route to a final destination router. Our tool implements an automated 1-hop indirect telnet scheme.

Sequencing Techniques

We present two main techniques for sequencing. The *treegen* method is a tree generation family of techniques. The *treegen* method deduces the routing topology to determine the order for updates. The *reachable* method is a sequential technique that uses indirect telnet (described in the previous section) in an interesting way to perform sequencing. For the following discussion, we refer to the technique that chooses to update routers in no particular order (i.e., effectively, randomly) as the heuristic random; unlike our techniques, the random technique does not aim to update all the routers.

The treegen Family of Techniques

The basic purpose of the *treegen* family of techniques is to deduce the routing path tree to the nodes that need to be updated, and perform the configurations bottom-up in this tree. Such a tree can be built by performing traceroutes to the routers that need to be updated. The output of the traceroutes can be put together (using commonly known techniques, e.g., as done in [2, 7] for a different problem) to obtain a directed tree of the routers which depicts how packets are routed to these routers.

A simple such traceroute tree is shown in Figure 3a; it may have been constructed through a traceroute from the network operations center (NOC) to router r_5 followed by traceroutes to routers r_2 and r_3 . The technique assumes that the part of the routing tree not yet updated will be stable for the time required to perform the configurations. Multiple routes to the same host may exist, but choosing one is generally sufficient.

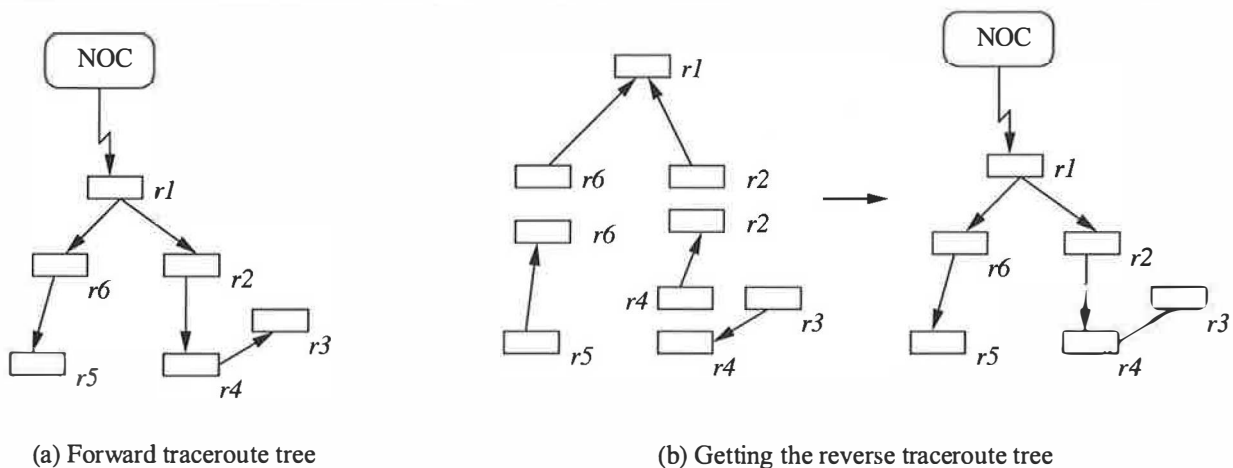


Figure 3: Forward and reverse traceroute trees.

Once the tree is obtained, the sequence is specified from the leaves to the root of the tree; i.e., all nodes reachable from a node must appear before the node in the sequencing order. Such an order can be obtained from a tree, e.g., by using depth first search techniques [1]. In Figure 3a, $r_3, r_4, r_2, r_5, r_6, r_1$, and $r_5, r_3, r_6, r_4, r_2, r_1$ are valid sequences. We call our technique described above as the traceroute method.

While doing a traceroute, if there is no response to a probe within the specified timeout, a "*" output is observed. It is not uncommon to see "*" outputs for all probes, especially at higher time to live values (i.e., for probes reaching farther from the host). This slows down the traceroute method and also makes it less reliable. Routers on the network with out-of-band connectivity could be exploited in the sequencing process. We now present some enhancements to our traceroute method that deal with some of these issues.

The reverse-traceroute Method

If we have access to the routers (i.e., the router passwords, which are needed for configuration), we can perform a traceroute *from* the routers to the management station rather than to the routers from the management station. These traceroutes can be put together to deduce the tree from the NOC to the routers. If the routing is symmetric, such a method that does traceroutes from the router would give the same tree as the generic traceroute method. In this case, however, we have the advantage that we can do a restricted traceroute, possibly looking only at the next hop on the routing path from the router to the management station. These {router, next hop} pairs can be put together to get the full tree. For the example in 3a, the method to obtain the reverse traceroute tree by conducting one-hop traceroutes from each of the routers is depicted in Figure 3b.

In theory, we need to examine the traceroute from each router until we hit a router that also needs to be updated. This ensures that we have the complete dependency between the routers that need to be updated. In most cases, however, the set of routers to be updated is contiguous and looking at the next hop suffices. If access to all routers exists, the routing table entries can be examined in lieu of a traceroute. A traceroute is a simple way to get the route information.

The forward-reverse Method

The safer method for constructing the tree is to perform both the traceroute and reverse-traceroute methods. This will avoid depending on the symmetric routing assumption. If we get a tree (or a directed acyclic graph [1]) by putting the trees derived from the traceroute and the reverse-traceroute methods together, the sequencing order can be determined. A cycle indicates a possible generic problem in determining a sequencing order. If such a situation is due to multiple routing paths in the network, there is no problem, since the routing path will readjust, if needed. If the situation is due to forced asymmetric routing (as described in

Figure 2), there is a problem. However, this is a pathological example and most installations do not design their networks this way.

Other Generalizations

There can be several possible start nodes for initiating the traceroutes. It is necessary to have access (i.e., login) to each of the routers and machines from which a traceroute is to be performed. Furthermore, there must be a path to the start nodes that does not go through other nodes that are to be updated. The following nodes are examples of candidates for start nodes: the machine in the network operations center, OSPF area border routers, and routers with out-of-band (e.g., modem or console) connections.

Let S be the set of nodes to which changes need to be made. Let SN be the non-empty set of possible start nodes. The output of the sequencing method is an ordered list of elements: (s_i, sn_j) , where s_i is a node from set S and sn_j is a node from set SN . The implication is that configuration changes must be made in the order specified by the list, and a telnet to node s_i must be accomplished from node sn_j (i.e., via an indirect telnet strategy).

Conceptually, we build *traceroute trees* from each of the start nodes. Each tree can either encompass all nodes in the network, or only the nodes not covered by already created trees. The dependencies can be computed from these sets of trees using standard graph theoretic approaches as in [1]; we omit these details from this paper. This generalization lends to interesting extensions of the basic approach, and techniques that mix pure in-band and pure out-of-band configuration approaches.

Optimizations

The treegen family of techniques lend themselves to various optimizations. Nodes that do not depend on each other can be updated in parallel to improve performance. The traceroutes in the reverse-traceroute method can be done in parallel to improve performance. In some cases, instead of deducing the routing topology by examining the routing tables or performing traceroutes, an analysis of the physical topology can indicate the routing topology (e.g., a simple line network).

The reachable Technique

Unlike the treegen family of techniques, the reachable technique merges the processes of determining the update order and the router configuration updates. At the i th iteration, the reachable technique determines the next router to update, updates this router, and then determines the $i + 1$ st router to update. The choice of the next router depends on the current state of the network. The reachable technique assumes that a one-hop indirect telnet access is available to reach a router.

The reachable algorithm updates reachable routers via telnet. If there are no remaining reachable routers, the process switches to indirect telnet via a

router whose neighbor is reachable. (Indirect telnet was explained in an earlier section.) More formally, let S denote the set of all routers to be reconfigured, let U denote the set of already reconfigured routers, let TBU denote the set of routers that remain to be reconfigured, and let R denote the set of routers that are currently directly reachable from the management station. Initially, $U = \Phi$, and $TBU = R = S$. The set R can be determined at any point in time via pings from the management station, and TBU is always equal to $S - U$. Algorithm *reachable* uses the following logic to determine the next router to reconfigure. If there is a router to be reconfigured that is directly reachable (i.e., in set R), it updates that router. If there is no directly reachable router (i.e., $R \cap TBU = \Phi$), *reachable* reconfigures via indirect telnet a router from set TBU whose neighbor is reachable (i.e., the neighbor is in set R). Otherwise, it declares failure. After configuring a router, it updates sets U and TBU , recomputes set R , and continues, as long as $TBU \neq \Phi$.

An interesting property of the *reachable* algorithm is that under most conditions, it will not fail. In other words, it will always find a router to update. We formalize this notion in Appendix 1.

Implementation

We implemented the techniques described earlier to obtain measurements on their efficacy. The implementation was done in Java. For our prototype version, we adapted a Java telnet implementation [6] based on Java sockets to contact the routers. The router passwords were available to log into the router and perform configuration. We wrote our sequencing routines as part of the IPNC tool [5, 13]. Our implementation allowed us to determine information about neighbors to the routers, and the ability to do one-hop indirect telnet, perform traceroute from the routers, generate the correct router commands and perform the required interactions with the routers.

The treegen Technique

We implemented the reverse-traceroute method from the *treegen* family of techniques. We implemented two variants of the method. In the first one,

which we call *trace-parallel*, after the order is determined, the routers are updated with maximum possible parallelism; in other words, at any stage, all leaves of the tree are updated in parallel. In the second variant that we call *trace-seq*, the routers are updated sequentially. In both cases, the traceroutes from the routers are executed in parallel to obtain the order of update speedily. We restricted ourselves to three hops of output from traceroute. The routers we used did not allow an easy (non-interactive) way of specifying the maximum number of hops, and this had to be worked in from our implementation.

The reachable Technique

Implementation of the *reachable* technique required indirect telnet support. We performed pings in sequence, and hope to move to a parallelized implementation soon. Coming up with a suitable timeout to account for routing stabilization is non-trivial; we approximated with retrying the configuration operation after waiting five seconds upon a failure, and this worked for us. We elaborate more on this later in a section on Running Time Measurements.

Experiments and Results

Experiments were conducted on networks of various topologies. Measurements on the three basic topologies (line, tree, and circle) shown in Figure 4 are reported below. The other topologies we experimented with were made up of these basic topologies, as are most networks. The main results from other topologies we tested were in agreement with what we present here.

The routers in our experiments were Cisco routers (25xx, 26xx, 36xx, and 40xx series), running various versions of IOS (all version 11.3 or higher). The interfaces were a mix of ethernet, high speed serial, serial, and token ring interfaces. For our experiments, we enabled the routers to run OSPF, put all routers in the same OSPF area, and changed the OSPF area authentication parameter on the routers. (See Appendix 2 for a brief description of OSPF.) In this case, a change at a router does not immediately make itself or other routers unreachable. The protocol takes

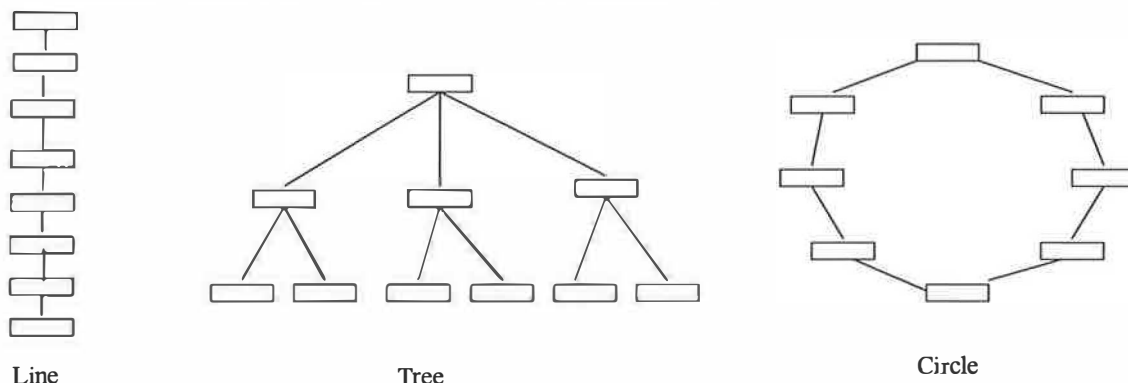


Figure 4: Network topologies for reported experiments.

some time to settle depending on other parameters (like hello and dead intervals) that we could change. Keeping the hello interval fixed at a low value (e.g., 1-2 seconds), a high value of the dead interval holds the network routing virtually unchanged for a long time, while a low value for the dead interval propagates the effect of the parameter changes quickly. For each of the methods random, trace-seq, trace-parallel, and reachable, we obtained measurements on (1) the number of routers that were successfully configured, (2) the time it took to determine the sequence order and (3) the time it took to perform the configuration. In the case of reachable since the sequencing is interwoven with the configuration, we only obtained one number: the total time to complete the reconfiguration. We now present our results under various categories.

Number of Routers Updated

The trace-seq, trace-parallel, and reachable techniques were always successful in updating all the

routers. This was true for all values of hello/dead interval we tested.

An interesting situation in this case is with the random technique. With random, since the routers are chosen in an arbitrary order and no sequencing is performed, some of the routers may fail to get configured.

Depending on the hello and dead intervals and the random order, up to 60% of the routers failed to get updated by random. An illustration of this effect is shown in Figure 5, which presents a plot for a random sequence. The plot in the figure must be taken with some caution. By using automated techniques rather than manual interaction with the router for configuration, quick router configurations were possible. (More details of the time for configuration appear below.) The effect from Figure 5 may be vastly different for manual configurations in that many more routers may fail to get updated, since each router takes longer to be updated when done manually. Second, the random

Method	Line (8 routers)			Tree (10 routers)			Circle (8 routers)		
	t_{seq}	t_{conf}	t_{total}	t_{seq}	t_{conf}	t_{total}	t_{seq}	t_{conf}	t_{total}
trace-seq	16.70	18.92	35.62	27.86	19.63	47.49	23.52	15.06	38.58
trace-parallel	18.16	17.81	35.97	27.87	6.14	34.01	19.83	9.82	29.65
reachable	—	—	58.92	—	—	69.46	—	—	56.05

Table 1: Average time to perform sequencing and configuration for the three topologies and the three techniques; t_{seq} is the time to determine the sequence, t_{conf} is the time to do configuration, and t_{total} is the total time taken. The times are for all routers in the network to be updated. All units are in seconds.

Method	Line (8 routers)			Tree (10 routers)			Circle (8 routers)		
	σ_{seq}	σ_{conf}	σ_{total}	σ_{seq}	σ_{conf}	σ_{total}	σ_{seq}	σ_{conf}	σ_{total}
trace-seq	2.31	0.52	1.79	3.43	0.41	3.35	0.06	0.05	0.11
trace-parallel	0.04	0.65	0.61	3.49	0.12	3.59	4.61	0.15	4.69
reachable	—	—	0.75	—	—	1.15	—	—	0.74

Table 2: Standard deviations in time to perform sequencing and configuration for the three topologies and the three techniques; σ_{seq} is the standard deviation in the time to determine the sequence, σ_{conf} is the standard deviation in the time to do configuration, and σ_{total} is the standard deviation in the total time taken. The times are for all routers in the network to be updated.

Method	Line (per router)			Tree (per router)			Circle (per router)		
	t_{seq}	t_{conf}	t_{total}	t_{seq}	t_{conf}	t_{total}	t_{seq}	t_{conf}	t_{total}
trace-seq	2.09	2.37	4.45	2.79	1.96	4.75	2.94	1.88	4.82
trace-parallel	2.27	2.23	4.50	2.79	0.61	3.40	2.48	1.23	3.71
reachable	—	—	7.37	—	—	6.95	—	—	7.01

Table 3: Average time (per router) to perform sequencing and configuration for the three topologies and the three techniques; t_{seq} is the time to determine the sequence, t_{conf} is the time to do configuration, and t_{total} is the total time taken. These numbers are derived from Table 1 by dividing the total time by the number of routers, and is presented here for convenience. All units are in seconds.

order is significant, and the plot implicitly says more about the time for the routing to settle rather than about the random technique itself. Third, the graph may look different if a parameter other than dead interval were chosen for the x axis. We chose the dead interval since it is a good control variable for how soon the configuration changes affect the routing.

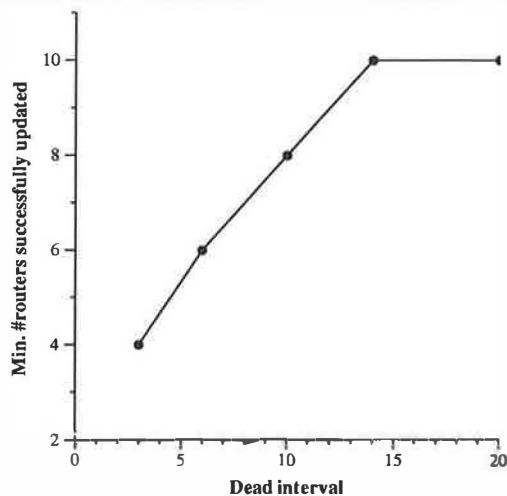


Figure 5: Minimum number of routers (out of 10) updated by random as a function of OSPF dead interval using a hello-interval of 1 second on the tree network.

Running Time Measurements

Tables 1-3 show various statistics related to the time taken by our techniques for determining the sequence and performing the configuration changes. All values are derived from 3-4 runs. (We did many more runs, but they used different topologies and different number of routers. We observed similar results from those experiments as reported here.) Table 1 shows the total time for all routers in the network to be updated and Table 2 shows the standard deviation of the observed times. For convenience, we divide the total time numbers from Table 1 by the number of routers in the network to get per router timing numbers in Table 3. The topologies have different number of routers (eight each for the line and circle, and ten for the tree); in addition, the routers and interfaces used in the topologies were also slightly different.

The first interesting result is that automation makes the total time taken to do the configuration very small (on the average, 0.6-2.23 seconds per router for trace-parallel as seen from Table 3). This enables rapid reconfiguration and makes our tool very useful. As a point of reference, our (informal) study shows that manual reconfiguration of authentication by experienced administrators may take about 30-45 seconds per router (*excluding* times for sequencing), resulting in 5-7.5 minutes for reconfiguring 10 routers. This estimate for manual configuration excludes the time to generate the configuration commands, but includes the time to type in the commands at the router's

configuration prompt. In the case of OSPF authentication, this typically translates to 4-6 commands per router. In the automated case, the commands are automatically generated and sent to the router by our tool, and that time is included in the reported numbers.

The automation also lends to easy parallelization of the configuration operation, when possible. The performance improvement resulting from the parallelization is visible from the t_{conf} value for the trace-parallel technique, which shows a 34.6% and 68.8% improvement over the corresponding t_{conf} time for the trace-seq technique for the circle and tree topologies, respectively. No parallelism is possible in the case of the line. The traceroutes in the case of the trace-seq and trace-parallel methods were done in parallel, and the t_{seq} time is dominated by the amount of time it takes to get a 3-hop output from the traceroute done at the router (since all other processing on this output is minimal and in-memory). For random, when it is successful in completing the configuration, the $t_{conf} = t_{total}$ and is approximately the t_{conf} value for trace-seq. From the table, it is difficult to break up the time taken for configuration alone by reachable. The time for non-configuration activities (including pings, timeouts, and indirect telnet) for reachable can be estimated, however, by

$$(t_{total}(\text{reachable}) - t_{conf}(\text{trace-seq})/t_{total}(\text{reachable}))$$
 to be 67-73% for the three topologies.

The comparatively low values for standard deviation from Table 2 suggest that the timing measurements did not vary significantly.

Interestingly, the timing values for trace-seq and trace-parallel were rather predictable and independent of the hello/dead interval values (and hence, the time taken for route changes). However, the same was not the case for reachable. The time for reachable depends on the order in which the routers are updated, and how soon the routes change. In our implementation of reachable, the order in which the routers were tried for update depended on the hash function used by Java (since the router names were initially put into a hash table). Figure 6 shows how the time for configuration for ignored: reachable changed with the dead interval in the network. The increase in time when the routes change more rapidly (i.e., at lower values of dead interval) comes from the increase in the number of indirect telnets performed and the timeouts when performing pings (when the routing is settling down). As mentioned in Appendix 1 and the implementation section, in our experiments, when there was a failure, retrying after a timeout of five seconds provided a router as per the condition in Proposition 1 in Appendix 1.

In rare circumstances (in two cases out of several tens of tests we ran), we observed that a ping to a router succeeded, but the routing changed between the ping's success and completion of the configuration. Such occurrences may be more frequent in some networks. However, it does not matter to the reachable

technique from the point of view of correctness, since it assumes that the router was unreachable. It does, however, increase the running time of the technique.

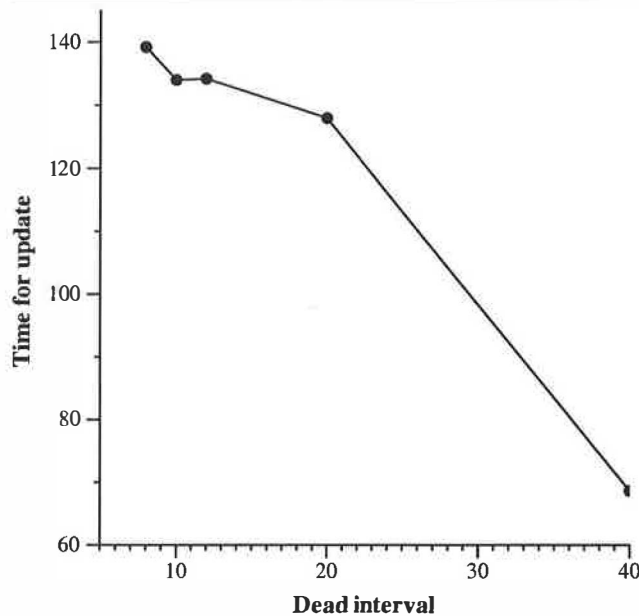


Figure 6: Time taken by reachable to do configuration as a function of the dead interval using a hello interval of 2 seconds on the tree network.

Asymmetry in Routing

In most of our topologies, the routing was symmetric. In the circle case, both paths around the circle could be taken by the packets. We did not explicitly manipulate the routing tables, and left the cost of both paths to be the same (for OSPF purposes). The dynamic nature of the routing combined with the sequencing techniques we used resulted in no configuration problems for the topology using our techniques. In our experimentation, we concentrated on topologies and scenarios that we felt were representative of real networks, and hence did not choose configurations like in Figure 2.

Conclusions

In this paper we have studied the problem of IP network configuration. In particular, for the first time, we have studied the issue of sequencing updates of critical parameters in a network of IP routers. We have presented several techniques to sequence the configuration of the routers to ensure continued connectivity to the routers during the time that it takes to perform the configuration. We have implemented our techniques, and have presented experimental results on the validity of the techniques and measurements of their performance. We have shown that automated configuration significantly speeds up time for configuration (from an estimated 45 seconds per router while done manually to 0.6-2.3 seconds when automated), and our sequencing techniques make configuration fast and reliable. Our automated techniques also make

the configuration process convenient for the administrator. Some of our sequencing techniques are also part of the IPNC tool [5]. We believe that this significantly advances the state of the art in a field dominated by manual configuration of routers. We believe that such tools and techniques for automated network configuration will prove valuable to the users.

The network configuration tool with appropriate enhancements is currently available from Lucent Technologies (<http://www.lucent.com>) and ISPsoft, Inc. (<http://www.ispsoft.com>).

Acknowledgements

We would like to thank Subrata Mazumdar in particular, and other members of the IPNC team at Bell Labs for helpful discussions.

Author Information

P. Krishnan (who is called "Krishnan" or "PK") is currently with ISPsoft, Inc.; pk@ispsoft.com. He received his Ph.D. in Computer Science from Brown University, and his B. Tech in Computer Science from the Indian Institute of Technology, Delhi. Prior to joining ISPsoft, he was with the Networking Research Center at Bell Labs, Lucent Technologies. His research interests include IP networking and management, the development and analysis of algorithms, prefetching and caching, and mobile computing.

Tejas Naik is currently with Bell Labs, Lucent Technologies Inc. He received his Master's degree in Computer Engineering from University of Southern California and B.E. in Electronics Engineering from South Gujarat University, India. Prior to joining Bell Labs, he was with QLogic Corporation. His research interests include IP network and services management and IP traffic management. Reach him electronically at tnaik@research.bell-labs.com.

Ganesan Ramu is currently with CoSine Communications Inc., Redwood City, CA. He received his B.S. in Computer Science and Engineering from University of Madras, Chennai, India. His prior work in Network Management area includes his association with the Networking Research Center at Bell Labs, Lucent Technologies and Network Management Business Unit, Cisco Systems. His current interest is in the analysis of various aspects of IP routing protocols implementation.

Roshan Sequeira holds a Bachelor's Degree in Electronics and Communications Engineering from Mangalore University, India. He has worked for the Network Management Business Unit at Cisco Systems and the Network and Services Management Department at Bell Labs. He is currently with ISPsoft Inc.

Bibliography

- [1] A. Aho, J. Hopcroft, J. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley Longman Inc., April 1978.

- [2] C. Cunha, *Trace Analysis and Its Applications to Performance Enhancements of Distributed Information Systems*, Ph.D. thesis, Boston University, 1997.
- [3] J. Case, M. Fedor, M. Schoffstall, and J. Davin, *A Simple Network Management Protocol*, IETF Network Working Group, STD 15, RFC 1157, May 1990.
- [4] J. Case, R. Mundy, D. Partain, B. Stewart, *Introduction to Version 3 of the Internet-standard Network Management Framework*, IETF Network Working Group, RFC 2570.
- [5] The IP Network Configurator, <http://www.lucent.com/OS/ipnc.html>.
- [6] M. Jugel and M. Meisner, *The Java Telnet Applet*, <http://www.first.gmd.de/persons/leo/java/Telnet/index.download.html>.
- [7] P. Krishnan, D. Raz and Y. Shavitt, "The Cache Location Problem," to appear in the *IEEE Transactions on Networks*.
- [8] D. Levi and J. Schoenwaelder, "Definitions of Managed Objects for Scheduling Management Operations," IETF Network Working Group, RFC 2591, May 1999.
- [9] D. Levi and J. Schoenwaelder, "Definitions of Managed Objects for the Delegation of Management Scripts," IETF Network Working Group, RFC 2592, May 1999.
- [10] John T. Moy, *OSPF: Anatomy of an Internet Routing Protocol*, Addison-Wesley Longman Inc., December 1997.
- [11] John T. Moy, "OSPF version 2," *Internet RFC 2328 Standard*, April 1998.
- [12] W. R. Parkhurst, *Cisco Router OSPF: Design and Implementation Guide*, McGraw Hill, July 1998.
- [13] B. Sugla and P. Krishnan, "IP Network Configurator: Advancing the State of the Art in IP Network Deployment and Operations," *Lucent Technologies' whitepaper*, November 1998.

Appendix 1: Success Criterion for the reachable Algorithm

The success criterion for the reachable algorithm is based on the following *communication property*. In most cases, nodes that have already been reconfigured,

and that are (topologically/logically) connected will start communicating with each other, and be reachable from each other. Given the correctness assumption from earlier, such nodes will continue communicating with nodes that do not need to be reconfigured and that are connected to them. Further, if any one of these nodes is reachable from the management station, all of them will be reachable.

While it is not essential that the communication property hold true, it often is, especially for routing protocol parameter updates. In such cases, the following proposition holds.

Proposition 1: Let the communication property be true. If there remain a set of routers to be reconfigured, there always exists a router in this set such that either that router or one of its neighbors is reachable.

Proposition 1 implies that when the communication property is true, algorithm reachable will always be successful in its reconfiguration operation. The validity of Proposition 1 is not hard to deduce; we present an informal proof below using Figure 7.

Let a be a type of node that does not need to be reconfigured (i.e., a node of type $a \notin S$), b_u be a type of node that has already been reconfigured (i.e., a node of type $b_u \in U$), and b_{tbu} be a type of node that has not yet been reconfigured (i.e., a node of type $b_{tbu} \in TBU$). (We use the set definitions from the earlier description of algorithm reachable for simplicity in exposition.) Take any node x from set TBU (i.e., of type b_{tbu}) and conceptually trace the routing path⁵ from the NOC to x . (We do not need to know this path; it is only used for the proof.) Map the nodes on the path to their types; for the example in Figure 7, we get a string of the form " $ab_ub_{tbu}ab_{tbu}$ ". By the communication assumption, the nodes corresponding to the initial string matching the regular expression " $(a*b_u)*$ " (in this case ab_ub_u) are reachable from the NOC, and hence a neighbor of the first b_{tbu} (i.e., the neighbor z of node y) is reachable. If the initial string " ab_ub_u " were empty, then the first node of type b_{tbu} is directly reachable. Note that it is not the node x we selected initially in our proof that is

⁵The term routing path is used loosely here. A formal proof would consider the path packets would take in trying to reach b_{tbu} , even if packets do not complete the journey.

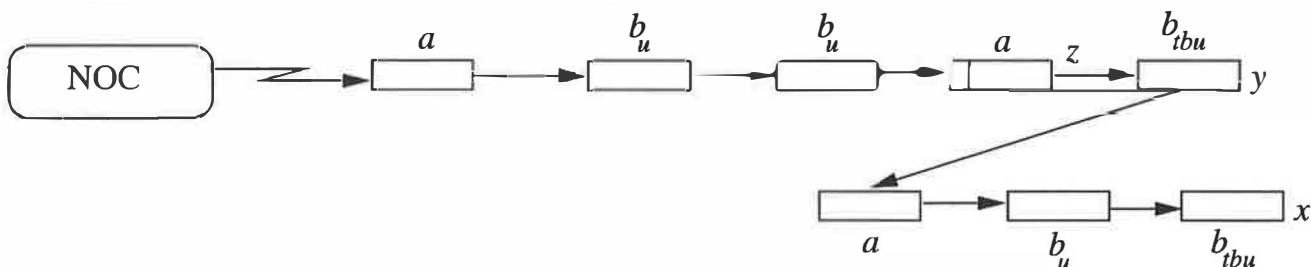


Figure 7: Reachability snapshot for Proposition 1.

reachable; all the proposition says is that there exists *some* node that is reachable or whose neighbor is.

The communication property does hold for the update of most routing protocol parameters. Even in such cases, the routing takes some time to settle down, and the communication property holds after the routing has settled down. The algorithm has to incorporate timeouts and retries to take care of this phenomenon. Typically, the time taken for the routing to settle down for a given network after each reconfiguration can be estimated to be some time t . If there are routers remaining to be updated, and we are unable to find any, retrying once after waiting for time t should provide a router as per Proposition 1. Although t can be pretty large in theory, in practice it is reasonably small, and the number of times you need to wait is expected to be infrequent. Another possible problem that could arise in practice is that a router is reachable but becomes unreachable before the configuration action is completed. We describe our experiences with these issues in time measurement section.

Appendix 2: Routing and OSPF – A Brief Overview

Routers are critical network elements of IP networks. The most important function of an IP router is to route data packets. A forwarding table inside a router is used to decide which route a packet should take. There are several ways to add entries to a forwarding table. Typically, a protocol builds this table dynamically. These protocols are called routing protocols. A routing domain is a collection of routers and all routers in a routing domain run the same routing protocol. One or more such routing domains constitute an Autonomous System (AS). A unique number identifies this AS to the rest of the autonomous systems. This number is called AS number. Routing protocols designed to run among AS are called Exterior Gateway Protocols (EGP). Routing protocols designed to run inside an AS are called Interior Gateway Protocols (IGP). One of the most popular IGPs is the Open Shortest Path First (OSPF) routing protocol [10, 11].

OSPF is a hierarchical routing protocol. It can run in the entire AS. The AS can be divided into OSPF areas. A unique number identifies each area. An area identified by number zero is also called the OSPF backbone. Grouping of routers in an area is a network planning and design issue, which is beyond the scope of this paper. A router is an Area Border Router (ABR) if it is part of more than one area. Entries in a forwarding table, i.e. routes, are generally aggregated at the area level by the ABR so that the processing burden and storage requirements of the routers in the other areas are reduced. For OSPF to work correctly, at least one ABR in an area must be connected to the OSPF backbone.

Some OSPF parameters used in this paper are described below.

- **Hello Interval.** A router running OSPF sends a “hello” packet periodically to inform its neighbors that it is alive. The fixed interval between consecutive hello packets is called the *hello interval*.
- **Router Dead Interval.** If a router does not receive a hello packet during a fixed time interval from a neighbor, it declares that neighbor dead. This interval is referred as the *router dead interval*.
- **Area Authentication.** A technique used to ensure that the router sending OSPF information is trusted and the OSPF packet has not been altered.

ND: A Comprehensive Network Administration and Analysis Tool

Ellen L. Mitchell, Eric Nelson, & David K. Hess – Texas A&M University

ABSTRACT

ND is a software tool developed by the Computing and Information Services Network Group at Texas A&M University (TAMU) to aid in the engineering and operation of the campus network. This tool was developed in response to the tremendous growth of the TAMU campus network over the last ten years. ND is designed to provide high-level application functionality while retaining the power and flexibility of a low level tool. It integrates remote network device analysis (via SNMP) with network defining databases (via SQL). ND is written in Python [1] and contains custom modules for interaction with SNMP and MySQL [2].

Introduction

ND is a tool that was developed to aid in the operation and, more importantly, the engineering of the TAMU campus network. With a network of more than 25,000 nodes and over 1000 network devices, a tool was required that could scale to a high level and provide useful functionality without the need to repeatedly develop special purpose scripting based solutions.

One of the major design goals of ND was to have a command line interface (CLI) syntax that was accessible remotely via Telnet/SSH. This is a very important feature because this type of remote access is the lowest common denominator that can usually be found on almost any host or network device in the field. It requires neither a graphical interface nor the installation of special software. In addition, the CLI syntax provides a common command structure independent of different vendors' software implementations due to the use of standard SNMP MIBs.

Another design goal was to integrate ND into the Unix environment in order to leverage and adopt some of the features of Unix that have given it the ability to scale. This integration also made it simple to use a number of freely available support tools and technologies such as Python, SNMP and SQL. The resulting synergies have resulted in a very powerful and useful tool.

This paper presents a background section on network management applications that explains why ND needed to be developed, a design section covering the architecture of ND, a section on how ND is used, and finally a section discussing what the resulting benefits have been.

Background

A quick search of the World Wide Web reveals that there are a remarkable number of packages available both commercially and as open source that purport to perform "network management". This leads

one to ask, "Why build yet another network management tool?"

Network management applications have evolved into four major categories: event management, performance management, policy management and finally "generic" network management. Event management applications are based on the monitoring and managing of network events such as the reception of SNMP traps and up/down state information based on ICMP echo request/replies (ping). Two tools in this category are Big Brother [3] and Micromuse's Netcool/Omnibus [4] product. Performance management tools collect, monitor, and present network performance information (usually collected via SNMP). InfoVista [5] and MRTG [6] are examples of tools in this category.

Policy management mainly refers to proprietary applications developed by network equipment vendors that attempt to manage complex network equipment functionality (such as quality of service and security) via a policy abstraction. Cisco and 3Com are examples of vendors that have developed tools of these types. Finally, the category of "generic" network management applications is the most recognizable one. Applications of this type tend to center around a Graphical User Interface (GUI), which displays information about network topology, network elements and status. Commercial versions of these tools typically incorporate an element manager, which provides a GUI that allows in-depth control of a network element. HP's OpenView [7] and Scotty/Tkined [8] are examples of this type of application.

One of the weaknesses of network management applications as a whole is that they tend to address only the operational aspect of "network management". Those who engineer networks tend to eschew these applications for their work because of their constrictive nature; while GUIs are useful for displaying complex information, they are typically implemented in such a way that limits the ability of users to perform complex operations on medium to large sets of

network elements. They also do not take into account nor provide support for the specific network management practices that may be in use at a particular site (naming conventions, addressing conventions, network layouts, port numbering patterns, organizational structure, etc.).

True to their nature, network engineers tend to focus on network management tools rather than network management applications as categorized above. These tools are typically programming/scripting languages (Perl, Python, Expect, etc.), SNMP MIB browsers, libraries and standalone tools (SNMP++, UCD Snmp, etc), and databases (MySQL). The result is that engineers build special purpose applications on a case-by-case basis that bridge this gap between the tools and the applications. This software tends to be powerful and effective but special purpose and difficult to maintain and extend.

ND was developed to specifically address this problem; it has been designed to be a powerful tool built on SNMP MIB functionality and SQL databases that provides a useful and friendly application interface. While the Simple Network Management Executive [9] (SNMX) package provides a powerful scripting and SNMP tool structure and is thus closest in nature to ND, ND is unique in the level of functionality it provides in the form of a flexible tool.

ND Architecture

Python

ND is written in Python and contains custom modules for interaction with MySQL and SNMP. Python was chosen for its ease of programming and its modular and object-oriented design. There are over 40 modules in ND (approximately 10,000 lines of code) grouped into 8 families. Table 1 summarizes the module families. The module families closely follow individual MIBs (both standard and proprietary). Several of the standard Python modules have been customized while another has been newly written. Table 2 summarizes the modified or created Python modules.

To assist in creating new modules, a Python class was written from which all modules are sub-classed. The parent class provides methods for parsing user input, redirecting output to pipes or files, providing a

command-line history, and providing a mechanism for methods common to all modules, e.g., help.

ND contains an extensive help system that is available at two levels. First, at each ND prompt, a question mark can be entered which will list all the commands available in that module. Second, the help command can be issued with a specific command which will cause the syntax and description of the command to be displayed.

Module	Description
mysql	Modified the initial connection code to use MySQL conf files
readline	Modified to allow for user defined history files
snmp++	A new module that allows Python access to the SNMP++ and UCD SNMP libraries

Table 2: Modified/new python modules.

Databases

ND makes extensive use of SQL tables and uses the MySQL server as a backend. Table 3 lists the major categories of SQL tables. Originally, ND used an early version of msql and db, which essentially had the ability to store simple flat files. To ensure that the database made sense from a higher network design perspective, ND was designed to keep databases consistent and free from typical types of errors as might be prone with manual input. This includes sanity checking of record relationships on the creation of new records and automatic generation of record id numbers (functionality that is now found in MySQL).

The Fiber Circuits and Twisted Pair Drops categories are the result of extensions that have been made to ND over time. These are accessed via the equivalent of small applications within ND that specialize in the management of the campus network's fiber optic plant and twisted pair wiring plant. This functionality was added to ND due to the CLI infrastructure ND provided and due to the usefulness of sharing this information with the rest of ND. Fundamentally, these small applications provide support for the processes used at TAMU related to plant management and demonstrate the ability of ND to support site-specific business practices.

Module Family	Description	MIB
fms	3Com FMS Repeater MIBs	Proprietary
ost	Alcatel Switching MIBs	Proprietary
bridge	Bridge MIB	RFC 1286 [10]
fr	Frame relay MIB	RFC 1315 [11]
repeater	Repeater MIB	RFC 1516 [12]
rmon	RMON MIB	RFC 1271 [13]
snmp	Basic SNMP Query	RFC 1213 [14]
netdb	Database applications	n/a

Table 1: ND module families.

Table Category	Description
Support	Various support data such as campus buildings and department descriptions.
Fiber Circuits	Defines fiber circuits
Twisted Pair	Defines each installed drop on campus.
Drops	
Device	Contains basic information about backbone devices

Table 3: Database architecture.

SNMP

SNMP is a fundamental aspect of ND [15]. Since Python contains no built-in support for SNMP, a new module was created. Initially, a set of functions that used the system method of the OS module was used to interface (externally as child processes) with the UCD SNMP [16] set of utilities. This proved inefficient and more importantly, resulted in a security risk since community strings would appear in the process table as arguments.

A search for lower-level tools was initiated and the SNMP++ [17] toolkit was discovered. This toolkit provides a very efficient interface into the SNMP GET, PUT and TABLE functions but it does not have any provision for parsing MIBs. For this reason, the interface to the UCD snmptool function was kept but an OID translation cache was added as one of the SQL tables. This cache all but eliminated the inefficiencies of using an external program.

It should be noted that the SNMP v1 protocol is insecure [18] by its nature since authentication tokens (community strings) are sent in plain text. Steps should be taken to guard against others snooping traffic on the network and discovering the community strings, which may provide the ability to change the configuration of a device. At TAMU, the campus network has been engineered in order to prevent eavesdropping.

Performance Issues

Since Python is used primarily to implement the user interface, the performance of ND is more closely tied to the performance of the underlying SNMP and MySQL toolkits rather than Python. It has turned out that the load on the MySQL database server, even with a network as large as TAMU's, has not been shown to be a problem.

The real bottleneck has turned out to be the network devices response times to SNMP queries. Depending on the CPU and memory of the network device, it may take tens of milliseconds to respond to an SNMP query. When large tables need to be traversed (which must be done serially under SNMP v1), this can result in a large delay before an ND command completes.

Another performance related issue is how ND reacts to poor network conditions. SNMP is a stateless

UDP protocol; reliable communications to a network device must be ensured by the SNMP software. Typically, when SNMP responses fail to return to an SNMP management agent, the agent will retransmit the SNMP request. Under poor network conditions, it is important to be able to control this behavior based on the desires of the user and the severity of the network conditions. ND provides timeout and retries parameters that control how long it takes for an individual SNMP request to timeout and how many timeouts are allowed before communication with the device is considered to have failed.

Command Structure

ND is a hierarchical, command-line interface program. We made this choice so that shell scripts could be written to automate complex and repetitive tasks. Also, this tool is frequently used in the field where the only available interface is a Telnet/SSH session.

The hierarchical nature of ND follows a logical progression that we have found assists in problem tracking and security incident investigations. For example, the FMS family contains six sub-modules: address, ports, security, traps, users, and misc. Each of these in turn contains between five and ten individual commands. Similar structures exist for Alcatel, RMON, and bridge MIBs.

The current position in the hierarchy is reflected in the ND prompt. When first invoked, ND starts at the root level. To navigate deeper into the hierarchy, the user needs only type the name of a module available at that level. The quit command is used to leave a module and to return to a higher point in the hierarchy. The '..' notation is used as a mechanism to reference another module's commands without leaving the current module. When a hostname is required in a command, the '!' notation can be utilized to reference the host used in the previous command. This is very helpful when the command is related to the same host but is different to the point that command line history editing is not very helpful.

The GNU readline toolkit, which provides command line editing and history, has been incorporated into ND via the readline module found in Python. It has been modified to generate user defined command line history files, which allow the history to be persistent between ND sessions.

There are a number of commands common to most ND modules: list, show, set, add, delete, enable, disable and help. These commands have basically the same syntax but are tailored to the functionality of the particular module. The commonality of the syntax provides a user with a good base understanding of how commands within any module work. They need only use the help command to find the specific syntax.

One of the most powerful features of the command line interface is the ability to pipe output of any

command to a Unix shell command or to a file. This is accomplished by using the normal shell characters of '|', '>', and '>>' at the end of an ND command. As an indication of this level of integration, ND does not have any output paging mechanism. The user is responsible for piping the output of any command to the paging program of their choice (e.g., more).

Using ND

All TAMU network operators and engineers use ND. The types of tasks that each group performs are quite different but ND is flexible enough for both. These tasks include troubleshooting, administration, monitoring, and maintenance.

Starting ND is as simple as typing `nd`. Before presenting the top level prompt of `nd>`, ND will execute any commands found in `~/.ndrc`. Typically, some top level set commands are executed in order to establish certain operating parameters that ND needs. The set command allows you to specify the following:

<code>sqluser</code>	user name to connect to MySQL server
<code>sqlpass</code>	password to connect to MySQL server
<code>sqlhost</code>	host where MySQL server is running
<code>timeout</code>	number of seconds for SNMP timeouts
<code>retries</code>	number of retries for SNMP queries
<code>snmplimit</code>	maximum number of rows to return in SNMP queries
<code>read</code>	SNMP read community string
<code>write</code>	SNMP write community string
<code>rows</code>	maximum number of rows to output before reprinting column headings
<code>debug</code>	sets the internal debug level for MySQL and SNMP
<code>historyfile</code>	sets the file to use for the history file between ND sessions
<code>echo</code>	toggles ND echoing user commands

Another command commonly found in `.ndrc` files is the `attach` command. The `attach` command allows simple tags to be defined for long or complex host names and also allows for community strings to differ from the default. For example, hostnames of networking equipment at TAMU follow a set pattern but are often tedious to repeatedly type.

Once in ND, the user need only type the name of a module to navigate deeper into the hierarchy and quit to navigate back up the hierarchy. The prompt

will change to indicate which module the user is currently in. Within each module the help command provides module and command specific help. There is no documentation or training material for ND other than the internal help system.

As mentioned previously, the ND command line interface is integrated with the Unix shell. Most commonly, users will need to take the output of commands and pipe them to an output pager of their choice. More experienced users will save the output to a file or pass it into a `grep` command looking for a particular pattern. Expert users will pass the output to `awk` or possibly even an external Perl or shell script. In support of this functionality, the `rows` parameter can be set such that no column headings are printed with a command's output.

Rather than give an exhaustive explanation of all ND modules and commands (which space will not allow), it is more useful to show by example, how ND is used at TAMU day-to-day. The following examples show how ND is used in a variety of both operational and engineering related situations.

The general format of common ND commands is:

```
<command> <host> <unit> <port> <options>
```

The units and ports can be specified using a scalar, list, or range notation.

Scenario 1: Ports 1, 3, 4, 5 and 8 of unit 1 of a new 3COM FMS have recently been activated. Before the users are informed that the ports are available, the configuration of the ports must be verified. The command shown in Figure 1 would be used.

Scenario 2: Ports have become scarce inside one building. It is suspected that some ports that are allocated are not actually being used. Viewing only the ports on unit 1 that have a total frame count greater than zero would indicate which ports are actually being used. The ND command and corresponding output are shown in Figure 5.

Scenario 3: Several users have called and complained that the network in their building has become unresponsive. It is suspected that someone is using too much bandwidth. By inspecting the frame counts for all ports on a device and then sorting the output, the ports using the most bandwidth are easily found. The output for this example is in Figure 6.

```
nd-fms-ports> show fms-device-1.domain.com 1 1,3-5,8 config
```

Unit	Port	Status	EST filter	Part. trap	Link trap	Link pulse	DUD	action
1	1	on	mac	enabled	disabled	enabled	N/A	
1	3	on	mac	enabled	disabled	enabled	N/A	
1	4	on	mac	enabled	disabled	enabled	N/A	
1	5	on	mac	enabled	disabled	enabled	N/A	
1	6	on	mac	enabled	disabled	enabled	N/A	
1	8	on	mac	enabled	disabled	enabled	N/A	

Figure 1: Verifying port configuration.

```

RMON Stats  Variable: Packets Received
              3              4
+-----+-----+
1 | 302414796 | 0
2 | 0         | 34631342
Help: ?
Host: host.domain.com
Mode: absolute
Rate: 0.5/sec Sample: 57

```

Figure 2: Dynamically monitoring RMON variables across all ports.

Scenario 4: A recent addition to ND has been the ability to continuously monitor a specific MIB variable across all units and ports of a network device. Python includes a Curses module that allows for simple screen control. This has been utilized to display a matrix of text where the columns represent slots or units, and rows represent ports. Figure 2 contains a single screen snapshot showing the RMON etherstats variable for the number of packets received. ND will query the device and update the screen every two seconds.

Scenario 5: There are over 500 3COM FMS type devices at TAMU. Our monitoring strategy is to have statistics that are summarized on a weekly basis. Clearing all the statistics on every FMS then becomes necessary. ND was specifically designed to be

incorporated into shell scripts so that fairly complex tasks can be accomplished easily. Figure 3 shows how all the statistics on all FMS devices on campus can be cleared at the same time.

```

(
echo "fms ports"
grep -v '^[%]' $FMSHOSTS |
while read host
do
    echo "clear $host of all statistics"
done
echo "exit"
) | nd

```

Figure 3: Shell programming with ND.

Scenario 6: The characteristics of all ports and units for a given device can be shown in a single table. This can be very time consuming when logging directly into a network device. Also, the device's software may not support tabular output of information. The ND command and resulting output for this example are shown in Figure 7.

Scenario 7: With over a thousand network devices at TAMU, new devices are constantly being added and old ones replaced. Using ND, a device can easily be configured to a known state. Figure 4 demonstrates how a set of specific ports can be configured easily using a shell script.

```

nd-fms-ports> show fms-device-1.domain.com 1 all counters | \
awk -- '{if ($7 > 0) print $0}'

```

Unit	Port	Util	Uni frms	Mult frms	Bcast frms	Tot frms	Ucast octs	Mcast octs	Bcast octs	Total Octets	Colls	Runts
1	2	0%	1286	0	11	1297	0	0	0	116943	2	0
1	3	0%	145846	0	5966	151811	0	0	0	20298744	209	0
1	6	0%	1638	0	764	2402	0	0	0	468142	0	0
1	7	0%	6697	0	1001	7698	0	0	0	531474	105	0
1	8	0%	16217	0	668	16885	0	0	0	1220390	16	0
1	10	0%	3268	0	18	3286	0	0	0	229193	1	0

Figure 5: Discovering which ports on a device are active.

```

nd-fms-ports> show fms-device-1.domain.com 1 1-10 counters | sort -k 4 -r -n

```

Unit	Port	Util	Uni frms	Mult frms	Bcast frms	Tot frms	Ucast octs	Mcast octs	Bcast octs	Total Octets	Colls	Runts
1	3	0%	205380	0	8453	213833	0	0	0	31735634	666	0
1	8	0%	23148	0	951	24099	0	0	0	1726534	63	0
1	7	0%	9413	0	1415	10828	0	0	0	746284	395	0
1	10	0%	4627	0	18	4645	0	0	0	324124	13	0
1	6	0%	2518	0	1090	3608	0	0	0	679204	184	0
1	2	0%	1803	0	17	1820	0	0	0	164001	6	0
1	9	0%	0	0	0	0	0	0	0	0	0	0
1	5	0%	0	0	0	0	0	0	0	0	0	0
1	4	0%	0	0	0	0	0	0	0	0	0	0
1	1	0%	0	0	0	0	0	0	0	0	0	0

Figure 6: Finding which ports on a device are using the most bandwidth.

Scenario 8: A customer calls and wants to know the speed of the workstation connections in his office. He provides the building and room number. The caller also wants to know the location of the 100 MB

connections in the building. Figure 8 shows the commands used to answer this query.

Scenario 9: A student in a residence hall has violated the acceptable use policy and is causing a

```
#!/bin/sh

# Find out which ports to configure as secure

snmptable -mPRODUCTMIB -R $TARGET public -H \
    mrmPortTable mrmPortCardIndex mrmPortIndex mrmPortInterfaceType |
    grep 'twistedPair' | sed -e 's/^ *\[0-9*\] *\[0-9*\].*/\1 \2/' |
(
echo set read commread
echo set write commwrite
echo fms security
while read PORT
do
    echo set $TARGET $PORT addresses 1
    echo set ! $PORT intrusion noAction
    echo set ! $PORT ntk NTKWithBcastAndMcast
    echo set ! $PORT mode continuousLearning
done
echo quit
echo exit
) | nd > /dev/null
```

Figure 4: Shell script for configuring a new device to a known state.

```
nd-fms-ports> show fms-device-1.domain.com 1 all diagnostics
```

Unit	Port	Total frms	Coll's	Runts	Colls/ frame	Runts/ frame	Late Frag	colls
1	1	0	0	0	-	-	0	0
1	2	2700	12	0	0.44%	0.00%	0	0
1	3	273442	916	0	0.33%	0.00%	0	0
1	4	0	0	0	-	-	0	0
1	5	0	0	0	-	-	0	0
1	6	4656	283	0	6.08%	0.00%	0	0
1	7	13857	620	0	4.47%	0.00%	0	0
1	8	30582	93	0	0.30%	0.00%	0	0
1	9	0	0	0	-	-	0	0
1	10	5947	23	0	0.39%	0.00%	0	0
1	11	815	8	0	0.98%	0.00%	0	0
1	12	8095	42	0	0.52%	0.00%	0	0
1	13	56349	2548	0	4.52%	0.00%	0	0
1	14	5	0	0	0.00%	0.00%	0	0
1	15	0	0	0	-	-	0	0
1	16	1574	4	0	0.25%	0.00%	0	0
1	17	30788	48	0	0.16%	0.00%	0	0
1	18	1222	66	0	5.40%	0.00%	0	0
1	19	0	0	0	-	-	0	0
1	20	790	0	0	0.00%	0.00%	0	0
1	21	21	0	0	0.00%	0.00%	0	0
1	22	0	0	0	-	-	0	0
1	23	2805	40	0	1.43%	0.00%	0	0
1	24	0	0	0	-	-	0	0
1	25	14047788	1032746	0	7.35%	0.00%	0	0
1	26	0	0	0	-	-	0	0

Figure 7: Displaying diagnostics for all units and ports in one command.

disruption. An engineer can disable the data ports to the room and schedule a meeting with the owner to discuss the problem. Figure 9 shows the commands used to solve this problem.

Benefits

One of the most important benefits that has resulted from the development and deployment of ND is the availability of a command line interface that is the same regardless of which type and make of network equipment is being accessed. This has always been the potential embodied in standard MIBs, though most network management application software has never taken advantage of it outside of an operational setting. The result has been that it is easier to train users. A module's commands will be effective on numerous devices regardless of the type and make of the device.

Another benefit that engineers have expressed is the ease of accessing ND and the simplicity of its interface. ND can be invoked from any Telnet/SSH session and is a system wide tool. No software needs to be installed in a user's account or on a workstation in the office or in the field. Also, the online help and commonality of the syntax between modules makes it easy to remember commands and learn new modules quickly. Command line history files and command line editing also make it simple to look up recent commands or use a previous command as the starting point for a new one. The result is that engineers are very efficient when using ND.

One of the seemingly mundane features but considered very valuable by the operators and engineers at TAMU is the ability to view a condensed, full page of statistics with a single command. This makes it simple for operators and engineers to get a picture of the overall health of a network device and quickly identify a problem. Usually, this is an action that would require a cumbersome series of queries when using the vendor's user interface.

The integration of the Fiber Circuits and Twisted Pair Drops database functionality has proven to be beneficial also. Leveraging the investment of learning ND and tying the information back to ND commands where appropriate not only saves human resources but has made the databases more valuable than they would otherwise be. In many environments, network databases of these types are considered a burden and are many times discarded because they have a different user interface from other tools and because the data is trapped in a separate system and not available in other contexts.

Another important benefit of ND is its ability to pipe output to a shell command or to a file. This feature gives users a powerful means of extending the functionality of ND in the traditional Unix style. Unlike many network management applications, the designers of ND realized that they could not foresee all the desires that operators and engineers would have and made ND extensible in this way. This has saved users from having to develop new special purpose applications and again saved resources.

Conclusion

ND was designed to be a powerful network management tool and to provide user-friendly functionality and extensibility. The Python language facilitates maintainable modules and a flexible command-line interface allowing the simple addition of new vendor-specific and standard MIB functionality as the network grows. The ND CLI eliminates the need for operators and engineers to learn complex vendor-specific syntax by providing a new interface based on the standard MIBs implemented by all vendors. Unix authentication and authorization mechanisms provide a basis for managing access to network devices independently of the individual device's authentication and authorization mechanisms.

It is clear that this type of approach to network management software is a powerful one. The

```
nd-netdb-drops> show room BLDG 311
```

The building and room number are entered.

Drop number	Type	Room	Length	DB loss	Dept.	Media	Connected to	Unit	Port
BLDG:1164	CAT5	311	106	0.0	DEPT	100BaseTX	dev1-ost5-1	4	7
BLDG:1019	CAT5	311	0	0.0		10BaseTX	dev2-fms2-1	3	6

```
nd-netdb-drops> show device dev1-ost5-1
```

One drop is 10MB, the other is 100MB. Check the 100MB device and see in which rooms the drops are terminated.

Device	Unit	Port	Drop number	Type	Room	Length	DB loss	Dept.	Media
dev1-ost5-1	4	1	BLDG:1136	CAT5	319	147	0.0	DEPT	100BaseTX
dev1-ost5-1	4	2	BLDG:1137	CAT5	312	157	0.0	DEPT	100BaseTX
dev1-ost5-1	4	3	BLDG:994	CAT5	011A	0	0.0	DEPT	100BaseTX
dev1-ost5-1	4	4	BLDG:1107	CAT5	011A	0	0.0	DEPT	100BaseTX
dev1-ost5-1	4	5	BLDG:1155	CAT5	107	91	0.0	DEPT	100BaseTX
dev1-ost5-1	4	6	BLDG:1064	CAT5	107	0	0.0	DEPT	100BaseTX

Figure 8: Helping a user Locate 100 MB ports.

development of ND began from the desire to build a friendly application that was also a powerful tool. A simple command line interface leveraging Unix was key to accomplishing this. More importantly, ND became a platform in which business practices could be embodied. At the same time, this embodiment can lead to weaknesses. Some business practices coded into ND may not be adoptable in other environments. They can even become a burden in TAMU's environment when it makes sense to change a business practice but ND would require significant redevelopment to support the change.

In the final analysis, every networking organization has a unique culture and a unique set of business practices. That organization must choose software that will support them in their mission. Tools that are too

low-level do not have interesting enough behavior for operators and most network management applications do not interest engineers because they are too restrictive or because they are not adaptable to the organization's business practices. Many times, the only perceived alternatives to this problem are to build completely custom network management applications or to change business practices. At TAMU, ND has successfully proven that simply designing a better tool is a viable alternative with many benefits.

Future Work

We are currently looking at integrating ND with an event management package. The event management package could trigger a lookup of a network device at a specific event threshold, execute a set of

```
nd-ost-vports> show dev1-ost3-1 4 12 The engineer checks the status of the port before disabling.
Slot Port Status VLAN          MAC          Prot  Encap Mode  Timer
-----
  4  12   on      1 00:20:aa:bb:23:12 TRN default auto   60
nd-ost-vports> disable dev1-ost3-1 4 10 The engineer disables the port and is then
                                         required to document this action.

Enter a comment to associate with this action: incident number 2000.143
nd-ost-vports> show dev1-ost3-1 4 10 Verify the status. Now a '*' reflects a comment is
                                         present for this port, and the status has changed to 'off'.

Slot Port Status VLAN          MAC          Prot  Encap Mode  Timer
-----
  4  10  * off    1 00:20:aa:bb:23:12 TRN default auto   60
nd-ost-vports> show dev1-ost3-1 4 10 comment If the student telephones the Operations Center before anyone
                                         can reach and discuss the problem with him, the Operations
                                         Center can check the status and inform the student of the situation.

Id          Host                Unit Port  Comment
-----
46294 dev1-ost3-1.net.tamu.edu 4    10 incident number 607.689
nd-ost-vports> enable dev1-ost3-1 4 10 When the problem has been resolved, the port is re-activated.

Enter a comment to associate with this action: incident 607.689 resolved
nd-ost-vports> show dev1-ost3-1 4 10
Slot Port  Status VLAN          MAC          Prot  Encap Mode  Timer
-----
  4  10  * on      1 00:20:aa:bb:23:12 TRN default auto   60
nd-ost-vports> show dev1-ost3-1 4 10 comment The comments are still assigned and can be retained, or
                                         can be removed with the 'comment delete' command.

Id          Host                Unit Port  Comment
-----
46294 dev1-ost3-1.net.tamu.edu 4    10 incident number 607.689
46295 dev1-ost3-1.net.tamu.edu 4    10 incident 607.689 resolved

Timestamp and audit information can be viewed using the
'comment info' command:
ID: 1          IP Address: 192.168.1.1  User: Operator #1 at server
Device: dev1-fms-1 Unit: 1          Port: 6
Date: 1999-10-19 Comment: disabled port for usage violations
nd-ost-vports> comment delete 46294-46295 Delete the comments by comment ID number:
```

Figure 9: Turning off a student's internet access.

predetermined ND commands, and generate a report for an operator to review and make a diagnosis. Additionally, while the Fiber Circuits and Twisted Pair Ports databases have a good command structure for querying them, it can be cumbersome populating the databases with bulk data. Additional work needs to be done on simplifying and bulletproofing this process.

Acknowledgments

We would like to thank the employees of the Texas A&M Network Group (Installation, Engineering and Systems teams) and to the Operations Center for being guinea pigs during the development phase and for patiently providing feedback to us on ND.

Availability

It has always been our intention to release the source code for ND. Extenuating circumstances prevented us from having the source available at the time of this writing; however, we expect to make it available at some time in the future.

Author Information

Ellen Mitchell obtained her Master's of Computer Science from Texas A&M University in 1994. While at Texas A&M, she has been the Systems Manager of the Computer Science Department and is presently a Security Analyst in the campus Network Group. She is involved in educating the campus user community concerning computer security. She may be reached by postal mail at Computing and Information Services, Teague Building, College Station, TX, 77843-3142 or by e-mail at ellenm@net.tamu.edu.

Eric Nelson has just recently left Texas A&M University where he was Systems Group manager of the campus Network Group. He has been the ND maintainer for the last several years and has developed many of the network analyst tools currently in use at TAMU. He received his Master's of Computer Science from Texas A&M University in 1987 and his Bachelors of Science from Texas A&M University in 1985. He may be reached by email at eric@net.tamu.edu.

David Hess was formerly the campus Network Manager at Texas A&M University. He is the original author of ND. David has extensive experience with network engineering and network management. He received his Master's of Computer Science from Texas A&M University in 1991. He recently left TAMU to join a startup company. He may be reached by e-mail at daveh@net.tamu.edu.

References

- [1] Lutz, Mark, "Programming Python," O'Reilly & Assoc., <http://python.org/>.
- [2] Yarger, Randy Jay, George Reese, and Tim King, "MySQL and mSQL," O'Reilly & Assoc., <http://www.mysql.com/>.

- [3] MacGuire, Sean, and Robert-Andre' Croteau, "Big Brother is Still Watching," SANS conference, Baltimore, Maryland, <http://bb4.com/>, May 1999.
- [4] Micromuse, Inc., "Netcool Suite Functionality and Benefits," <http://www.micromuse.com/products/overview.html>.
- [5] InfoVista, "The Challenge of Service Level Management," InfoVista Marketing, <http://www.infovista.com/products/frproducts.html>, 2000.
- [6] MRTG, <http://ee-staff.ethz.ch/~oetiker/webtools/mrtg/mrtg.html>.
- [7] HP OpenView, <http://www.openview.hp.com/>.
- [8] Schönwälder, J., and H. Langendörfer, "How to Keep Track of Your Network Configuration," *Proceedings, 7th Conference on Large Installation System Administration (LISA VII)*, Monterey, California, <http://wwwhome.cs.utwente.nl/~schoenw/scotty/>, November 1993.
- [9] Davison, Jeff, "Network Management Automation Using ACE Automated Console Expert," Diversified Data Resources, Inc., <http://www.ddri.com/Products/ace-snmx.html/>.
- [10] Decker, E., P. Langille, A. Rijsinghani, and K. McCloghrie, "RFC1286: Definitions of Managed Objects for Bridges."
- [11] Brown, C., F. Baker, and C. Carvalho, "RFC1315: Management Information Base for Frame Relay DTEs."
- [12] McMaster, D., and K. McCloghrie, "RFC1516: Definitions of Managed Objects for IEEE 802.3 Repeater Devices."
- [13] Waldbusser, S., "RFC1271: Remote Network Monitoring Management Information Base."
- [14] McCloghrie, K., and M. Rose, "RFC1213: Management Information Base for for Network Management of TCP/IP-based internets: MIB-II."
- [15] Stallings, William, "SNMP SNMPv2 and RMON," Addison Wesley.
- [16] UCD SNMP, <http://ucd-snmp.ucdavis.edu/>.
- [17] Mellquist, Peter Erik, "SNMP++ Specification," Hewlett-Packard Company, <http://rosegarden.external.hp.com/snmp++/index.html>.
- [18] Galvin, J. M., K. McCloghrie, and J. R. Davin, "Secure Management of SNMP Networks," *Proceedings of the IFIP TC6/WG 6.6 Second International Symposium on Integrated Network Management*, pp. 703-714, North-Holland, 1991.

Combining Cisco NetFlow Exports with Relational Database Technology for Usage Statistics, Intrusion Detection, and Network Forensics

John-Paul Navarro, Bill Nickless, & Linda Winkler – Argonne National Laboratory

ABSTRACT

Argonne National Laboratory operates a complex internal network with a large number of external network peerings. A requirement of this network is that it be monitored with minimal impact on traffic. Cisco NetFlow technology provides the information necessary to monitor such a network, but the data from NetFlow must be captured and analyzed. We present a system that uses a high-powered relational database to manage the data. Our primary motivations in building this system were to learn whether or not database technology was an appropriate tool for this situation and to understand what types of questions about the network could be answered with such a system.

This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, U.S. Department of Energy, under Contract W-31-109-Eng-38.

The Problem

High Performance Network with Minimal Firewall

Argonne National Laboratory peers with more than 50 external Internet Service Providers and Internet2 networks, at rates up to OC-12 (622 million bits per second). Soon we expect those peerings to increase in speed to Gigabit Ethernet and higher. Argonne's networks are based on Asynchronous Transfer Mode (ATM) and switched 10/100/1000 Megabit/second Ethernet. Argonne's network has two separate border routers that handle these peerings. While these border routers can take over for each other in the case of a failure, normally they do not see each other's traffic.

This is an interesting situation for a number of reasons. First, there is no single point in our network that can be used to monitor all traffic into and out of the lab. Second, the external network speed is a critical issue to a number of research groups in the lab – slowing down the networks by running them through filtering or monitoring routers can have a huge impact on many experiments. And finally, the number of peerings and their exact topology frequently change as we modify our external networking relationships.

What's Going On in the Network?

We needed a way to examine network utilization statistics, perform basic intrusion detection, and look at traffic patterns. The obvious way to do this would

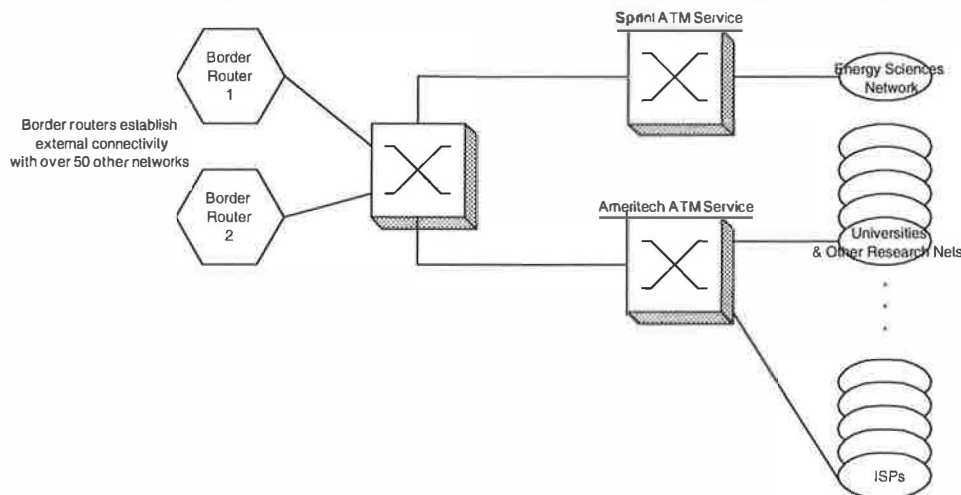


Figure 1: Router map.

be to drive all traffic through a router and use the router to watch every packet, but we are unwilling to create an artificial bottleneck in our network for network statistics gathering or intrusion detection purposes.

Unfortunately, many available network statistic gathering and intrusion detection systems, such as Network Flight Recorder, require that the intrusion detection system be able to inspect each packet as it enters and leaves the network. The Argonne network is not amenable to general packet inspection even at some small number of points for the reasons outlined above.

Thus we needed to find some way to keep an eye on our network traffic without measurably impacting its performance.

The Scenario

Fortunately, we had other options to explore.

Cisco has a technology known as "Netflow", which provides a summary of traffic through a router. We also have some significant experience with databases, which led us to believe that a database would be the optimal way of tracking Netflow data from multiple routers. Initially, we weren't sure how much data would be generated from our network, but, since we operate a number of supercomputers to support scientific experiments, we felt that we probably had the computing resources on hand to at least study the situation.

The motivating question for us was this: Is database technology a good way of tracking and analyzing NetFlow data?

Cisco Netflow Overview

Here is how Cisco describes this technology: A network flow is defined as a unidirectional sequence of packets between given source and destination endpoints. Network flows are highly granular; flow endpoints are identified both by IP address as well as by transport layer application port numbers. NetFlow also utilizes the IP Protocol type, Type of Service (ToS) and the input interface identifier to uniquely identify flows [1].

The IP routers in our network do NetFlow switching already, as it is a very efficient way of enforcing Access Control Lists (ACLs). The first packet of a flow will be inspected to see if it should be permitted or denied. Once that determination is made, a flow record is placed in the router forwarding cache. Following packets are matched against the flow record and are forwarded appropriately. Once the flow record cache line times out, a UDP/IP packet is generated and forwarded out of the router to a collection station.

The resulting NetFlow records provide a summary of just about anything a network statistics

collector or intrusion detection system may want to know about the traffic, with the exception of the actual packet contents themselves.

Database Technology

We run a number of Oracle and MySQL database servers to support our researchers. One of the authors (Navarro) was a professional Database Administrator (DBA) for a large Oracle installation at a previous employer.

Over the past several years, we have moved many of our critical infrastructure applications such as our list of hosts, lists of users, and so on, from flat files or simple lists into relational databases. Thus, many of our staff are becoming experienced with using databases for systems and network administration.

Thus we were fairly confident that a database would be the right solution for storing NetFlow data and that we would have the ability to manipulate it as we wished. The remaining question was – what should we use for a database server?

High Powered Database Server

We support a collection of workstations, clusters, and supercomputers as well as our networks. Our existing database servers run primarily on small Linux boxes or Solaris machines, and we knew that none of them would be up to the task of keeping up with the NetFlow data. The only spare computers we had were obviously underpowered.

One of the supercomputers that we support is a 96-processor SGI Origin 2000 [2] with approximately 2 terabytes of Fibre Channel disk. This machine is primarily used to support computer science and computational science. So, we decided to run this project as an experiment and, therefore, to justifiably use the Origin 2000 to run the database. We were fairly confident that it would have sufficient computing power to keep up with the NetFlow data during our initial tests, after which we could more accurately predict what kind of system would be necessary for production.

The Creation

Here, we describe the system that we built to capture and analyze the router NetFlow data.

Netflow to Database on Origin 2000

There are three basic parts to our configuration: the actual routers generating NetFlow data, a Perl [3] script to catch the NetFlow data from the network, and the back-end SQL database running on the Origin 2000. We experimented with Oracle 8i [4] and MySQL [5] back-end SQL software. We used on-line data capture where incoming NetFlow data went into the database directly. Our Perl script that caught the NetFlow data from the network would simply make DBI [6] calls to insert the data into the back-end SQL database. The primary advantage of this method is that the database is updated in real time, but has the

disadvantage that data can be lost if the back-end SQL database is unavailable for any reason.

We also used off-line data capture where the incoming NetFlow data was written to a temporary disk text file and then periodically loaded in bulk to the back-end SQL database. This allowed us to continue capturing data while the back-end SQL database was unavailable. It also allowed us to insert the data into multiple back-end SQL database engines when we wanted to compare their performance.

Database Schema Design

We chose a very simple database schema, drawn directly from the NetFlow specification itself. Our database consists primarily of a single table. Each row in the table is a single NetFlow record. Each column in the table has a one-to-one correspondence with NetFlow fields. We added only one extra column in the table to identify which border router generated the flow record.

```
create table netflows (
  router_id char(1) not null,
  src_ipn bigint unsigned not null,
  dst_ipn bigint unsigned not null,
  nxt_ipn bigint unsigned not null,
  ifin smallint unsigned not null,
  ifout smallint unsigned not null,
  packets integer unsigned not null,
  octets integer unsigned not null,
  starttime timestamp not null,
  endtime timestamp not null,
  srcport smallint unsigned not null,
  dstport smallint unsigned not null,

  tcp tinyint unsigned not null,
  prot tinyint unsigned not null,
  tos tinyint unsigned not null,
  srcas smallint unsigned not null,
  dstas smallint unsigned not null,
  srcmask tinyint unsigned not null,
  dstmask tinyint unsigned not null
)
```

This schema is so simple because we did not want to hinder our ability to make queries across the data later. We were interested to see how far modern SQL database engines could take us without optimizing the tables for particular types of queries.

Using The System

The system as described above is actually the complete system that we are using at present. We created the infrastructure to get the data from the routers into the database and then left it at that. Future steps (as discussed below) might be to provide nice interfaces to the data in the database, but this wasn't our initial goal. Rather, we wanted to see what kinds of questions could be answered with the data. From this point on, we've experimented with the system by forming basic SQL queries and examining the results. Our simple database scheme was designed with exactly this usage in mind.

Once you have the NetFlow records stored in the relational database, you can run many different types of queries against them. Here, we present some of the ad-hoc queries that we've found that we frequently use.

Usage Statistics

This is an example SQL query for someone interested in network statistics, perhaps asking the question "what Autonomous Systems (ASes) have we exchanged the most amount of traffic with recently?"

```
# Give a list of interesting
# ASes (high traffic flow)
select srcas,dstas,sum(octets) as bytes
from netflows
group by srcas,dstas
having bytes > 10000000
order by bytes
```

The resultant table lists each pair of Source AS and Destination AS that we have exchanged traffic with,

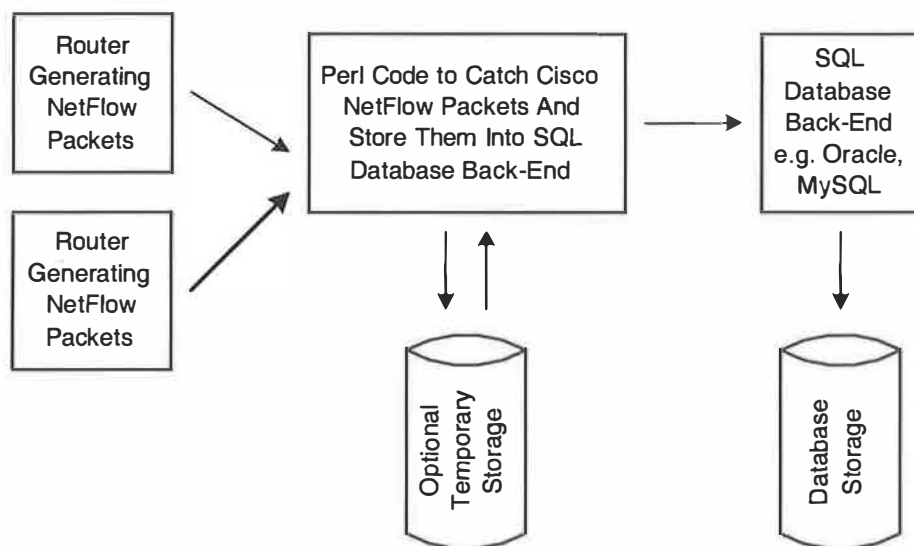


Figure 2: Database layout.

including our own, with more than 10 million bytes throughout the data stored in the NetFlows table, with the Source AS/Destination AS pairs that exchange the most data at the top.

Intrusion Detection

A network security officer might be interested in what outside IP addresses are trying to perform scans. Here is a query that illuminates the answer to that question:

```
select src_ipn, count(distinct dst_ipn)
                        as num_anl_addrs
from netflows
where srcas > 0 and starttime >
      date_sub(now(), interval 3 day)
group by src_ipn
having num_anl_addrs > 64
order by num_anl_addrs
```

This query gives the network security officer a list of IP addresses from external ASes that have contacted a large number of internal IP addresses. Given an IP address high on this list, the security officer might want to see what that IP address has been doing:

```
select src_ipn, min(starttime) as first,
              max(endtime) as last,
              count(distinct dst_ipn) as
                  addrs, prot, dstport
from netflows
where src_ipn = 140221009006
group by prot, dstport
order by addrs
```

ICMP Echo Reply (ping) scans, port scans, and even NMAP stealth scans show up very obviously in this and the preceding report.

Network Forensics

Let's say that the network security officer suspects that a machine on her network has been compromised. The officer would like to go back in time to see where any network connections came from, what kind of network protocols were used to compromise the host, and see where any outgoing connections may have gone. The officer may even wish to look at other

machines on the same subnet. Here is one query that might be used to do this; see Listing 1. The officer then gets a list of all the traffic entering or leaving a network between two given points in time. Such a report can be used to narrow down the type of attack that might have been used, and whether the compromised machine(s) on that network were used to attack hosts elsewhere.

Results and Practical Applications

Sizing It: Performance and Resource Impact

Not everyone has a spare 96-processor Origin 2000 sitting around to host a NetFlow database. (Not even us, really.) Thus, one has to ask questions about the investment necessary in time, computer hardware, and software to appropriately host a NetFlow database.

The first variable to consider is the rate of NetFlow records generated by your routers. This rate will vary widely depending on your usage, whether your network is being actively scanned, and the time of day. The best way to get a handle on this question is to simply go ahead and install the Perl code that catches NetFlow packets and saves them as flat files. Run this for a few days, preferably during the workweek as well as over a weekend. You can then run simple line counting utilities over these flat files.

Our routers generated about 14 million flow records on Monday, July 24 2000. Our Perl script stores the flow records in 5-minute batches. These batches ranged from 26,564 to 80,634 records per batch.

It is important that your database engine can accept table inserts much faster than your routers can generate NetFlow records. You need some performance overhead to handle queries, outages, and timeouts. We recommend trying to choose a SQL back-end system that can handle at least triple your real-time NetFlow record generation.

Using our example from July 24, we would need a SQL back-end database that can handle about 1500

```
# Something very weird happening at a particular time on a particular net.
select min(starttime) as startt, max(endtime) as endt,
       src_ipn, dst_ipn, prot, srcport, dstport,
       sum(packets) as pkts
from netflows
where ( ( starttime > 20000105003200
        and starttime < 20000105003600 )
      or ( endtime   > 20000105003200
        and endtime  < 20000105003600 ) )
group by src_ipn, dst_ipn, prot, dstport
having ( src_ipn >= 146137000000
        and src_ipn <= 146137255255 ) or
       ( dst_ipn >= 146137000000 and
         dst_ipn <= 146137255255 )
order by pkts, src_ipn, dst_ipn, prot, srcport, dstport
```

Listing 1: Examining other machines on the same subnet.

insertions per second at a bare minimum. Preferably we would want a database that could handle triple that rate, or just under 5000 insertions per second.

The next variable to consider is how long you want to keep NetFlow records around. This dictates your overall storage requirements. Be aware that the size of your NetFlow table may impact the speed of table inserts, which feeds back into the performance requirement discussed above.

We chose to keep about two weeks' worth of data on line. This time frame is a compromise between our desire to protect the privacy of our network users and having the data around to look into unusual events. To protect the long-term privacy of our network users we specifically chose not to back up the database itself.

Turnaround Time vs. Insertion Performance

You will also choose which table columns to index, if any. This is a tradeoff between insert performance and query performance. If you intend to simply capture data and look at it when something bad happens, you may not wish to spend the cycles indexing data that will be thrown away. The downside is that every query will require a full scan through the data before it can return results.

Some queries just about require a full table scan, such as the Intrusion Detection example query we provide above. If you anticipate doing those types of queries on a regular basis, you might choose to buy the fastest disk system you can find and not bother doing any indexing at all.

Database Technology Choice Makes a Big Difference

As mentioned above, we tried MySQL and Oracle on the SQL database back-end host. As part of their license agreement, Oracle doesn't let you publish specific performance numbers. But we can make some general comparisons between the two systems.

Oracle costs money. MySQL is (pretty much) free.

Oracle supports transactions, which slows it down but protects the database against corruption if the underlying host crashes for some reason. If the host system crashes under MySQL, you often have to re-index the table completely, which can take a very long time.

The MySQL query optimizer is sometimes less sophisticated than we might wish. Consider the query in Listing 2, assuming that the `src_ipn` and `dst_ipn` columns are indexed. This query took about 8 minutes to run, returning 576 records. The MySQL engine did a full 175,460,008-record table scan looking for matching rows.

However, the slightly simpler query in Listing 3 took 0.04 seconds to return 16 records. This time the MySQL engine used the `src_ipn` index to find the matching records much more efficiently than doing a full table scan. (The other obvious query using the `dst_ipn` index took about five seconds and returned the other 560 rows.) A more sophisticated query optimizer than MySQL provides would have split the original query into the two much simpler-and faster-queries and then combined the results.

If you've already decided to use MySQL, obviously you will want to run the two fast queries instead of the one slow query. But if you haven't yet decided on a SQL database back-end system, you may wish to run some sample queries and see how the optimizer treats each of them. The query optimizer is also critical to achieving good performance in a SQL database that supports parallel queries.

Other database products may provide features that could be used to speed up certain queries. Consider the Usage Statistics example query. If one were to drop the time constraint, it might be a perfect fit for an IBM DB2 [7] Summary Table. That is, the statistics would be generated "on the fly" during the NetFlow record insert process.

```
select src_ipn,dst_ipn,prot,srcport,dstport,
       sum(octets) as bytes, min(starttime) as first,
       max(endtime) as last
from netflows
where src_ipn=146139112126 or dst_ipn=146139112126
group by src_ipn,dst_ipn,prot,srcport,dstport
order by bytes
```

Listing 2: One form of query.

```
select src_ipn,dst_ipn,prot,srcport,dstport,
       sum(octets) as bytes, min(starttime) as first,
       max(endtime) as last
from netflows
where src_ipn=146139112126
group by src_ipn,dst_ipn,prot,srcport,dstport
order by bytes
```

Listing 3: Quicker query.

Futures and Conclusions

Non-Supercomputer Database Hosts

We cannot dedicate our 96-processor Origin 2000 to this application. So we must choose the most efficient combination of disk drive(s), CPU, memory, and database software to meet our NetFlow record generation rate and query types.

We may even choose to run several different SQL database back end servers, each optimized for a particular type of query.

We are still in the process of deciding exactly what system to use for this. As usual, it comes down to more of a question of budget than necessary technology.

Higher Level Tools Necessary For Non-DBAs

We are fortunate to be a cross-disciplinary team of network professionals, system administrators, and database administrators. Each of us is comfortable with writing simple ad-hoc SQL queries to look at the NetFlow data. Over time we hope to build higher-level tools that could be accessible to a wider audience. Examples might include a web interface that network security officers could use to view the activities of a given host, email alerts based on intrusion detection queries, or even graphics of network utilization over time.

Database Technology Is a Good Solution

The good news is that the higher-level tools can be built using the wealth of tools available for SQL databases. There is any number of commercial and/or open-source systems available to make web pages that use SQL database back ends. Just about any report-writing program can take data from a SQL query.

Database technology provides a good abstraction for this problem. It separates the nitty-gritty details of data storage from the actual problem we are interested in solving. As database software technology improves we get the benefits of those improvements right away. We are not stuck with a particular architecture tuning that limits us from following the Moore's Law increases in processor and disk storage performance. As our needs increase, we can swap out an under-performing SQL back end system entirely, as needs and budgets permit. Finally, we can take advantage of all the data protection and administrative expertise of an existing SQL infrastructure.

Availability

All of the code we used in this work is written as Perl and Bourne Shell scripts. There are probably no more than about 200 lines of actual source code. Look for a pointer to the NetFlow Database Perl Code on the <http://www.mcs.anl.gov/systems/software/> web page by the time this paper is presented at LISA.

Author Information

John-Paul Navarro <navarro@mcs.anl.gov> has been working at Argonne National Laboratory for 3

years. His professional interests include Linux clusters, scalable cluster management, and relational databases.

Bill Nickless <nickless@mcs.anl.gov> is a 9-year veteran of Argonne National Laboratory. His professional interests include high performance networking and data storage. He is never more happy than when fine tuning BGP policies while putting bar code labels on magnetic tape cartridges.

Linda Winkler <winkler@mcs.anl.gov> is Senior Network Engineer at Argonne National Laboratory's Mathematics and Computer Science Division. She also serves as MREN Technical Director and is a member of the STARTAP engineering team. Her focus since 1995 has been in the area of interconnectivity and interoperability of wide-area research networks in support of advanced scientific and engineering applications.

References

- [1] http://www.cisco.com/warp/public/cc/cisco/mkt/ios/netflow/tech/napps_wp.htm.
- [2] <http://www.sgi.com/origin/>.
- [3] <http://www.perl.org>.
- [4] <http://www.oracle.com/database/oracle8i/>.
- [5] <http://web.mysql.com/>.
- [6] <http://search.cpan.org/doc/TIMB/DBI-1.13/DBI.pm>.
- [7] <http://www.ibm.com/db2>.

The OSU Flow-tools Package and Cisco NetFlow Logs

Mark Fullmer – OARnet
Steve Romig – The Ohio State University

ABSTRACT

Many Cisco routers and switches support NetFlow services which provides a detailed source of data about network traffic. The Office of Information Technology Enterprise Networking Services group (OIT/ENS) at The Ohio State University (OSU) has written a suite of tools called flow-tools to record, filter, print and analyze flow logs derived from exports of NetFlow accounting records. We use the flow logs for general network planning, performance monitoring, usage based billing, and many security related tasks including incident response and intrusion detection. This paper describes what the flow logs contain, the tools we have written to store and process these logs, and discusses how we have used the logs and the tools to perform network management and security functions at OSU. We also discuss some related projects and our future plans at the end of the paper.

NetFlow Accounting Records

We should start with a more complete description of what the flows are. Quoting from Cisco:

A network flow is defined as a unidirectional sequence of packets between given source and destination endpoints. Network flows are highly granular; flow endpoints are identified both by IP address as well as by transport layer application port numbers. NetFlow also utilizes the IP Protocol type, Type of Service (ToS) and the input interface identifier to uniquely identify flows [3].

A NetFlow record is created when traffic is first seen by a Cisco router or switch that is configured for NetFlow services. Flows are identified uniquely by characteristics of the traffic that they represent, including the source and destination Internet Protocol (IP) address, IP type, source and destination Transmission Control Protocol (TCP) or User Datagram Protocol (UDP) ports, type of service and a few other items. NetFlow records end and are sent to the logging host on at least the following conditions:

- For flows representing TCP traffic, when the connection is done (after a RST or FIN is seen)
- When no traffic for the flow has been seen in 15 seconds.
- 30 minutes after the start of the flow. This causes long lasting traffic patterns to show up sooner than they might otherwise in the log.
- When the flow table fills.

Each NetFlow record contains data about the packets that are represented in that flow in addition to the unique identifiers listed above. These data include the start and end times for the flow, the number of packets and octets in the flow, the source and destination Autonomous System (AS) numbers, the input and output interface numbers for the device where the NetFlow record was created, the source and destination net masks and, for flows of TCP traffic, a logical 'or' of all of the TCP header flags seen (except for the ACK flag). In the case of Internet Control Message Protocol (ICMP) traffic, the ICMP type and subtype are recorded in the destination port field of the NetFlow records.

For example, suppose that a SSH connection is established from a client on host 128.146.222.233 port 1234 to a server on host 131.187.253.67 port 22, and that the traffic passes through a Cisco device that has NetFlow processing enabled. We will simplify things and identify our flows here by a tuple containing the IP Protocol type, source IP address, source TCP/UDP port, destination IP and destination TCP/UDP port. The initial packet from the client to the server causes the router to create a flow entry for {TCP, 128.146.222.233, 1234, 131.187.253.67, 23}. The response from the server to the client causes the router to create a related flow {TCP, 131.187.253.67, 23, 128.146.222.233, 1234}. Data from subsequent traffic will be aggregated in these two flow records until one of the ending conditions listed above is seen, such as

```
tc4>show ip cache 131.187.253.67 255.255.255.255 flow
```

SrcIf	SrcIPAddress	DstIf	DstIPAddress	Pr	SrcP	DstP	Pkts
AT2/0.31	128.146.222.233	AT3/0.1	131.187.253.67	06	03FA	0016	4
AT3/0.1	131.187.253.67	AT2/0.31	128.146.222.233	06	0016	03FA	8

Figure 1: Active flows as seen on a Cisco router.

when the TCP session ends, or because there has been no traffic for 15 seconds. Active flows can be viewed in the router command line interface with the command `show ip cache <IP Mask> flow`. This allows you to view flows that exist on the router whose NetFlow records have not been exported yet (Figure 1).

In the simplest case for a TCP session there will be a single flow representing the traffic from the client to the server, and a single flow representing traffic from the server to the client. The TCP flags field for both flows would typically have both the SYN and FIN bits set, indicating that packets with those flags had been seen traveling in both directions.

This is not typical, however. Traffic for a single TCP connection is frequently represented by multiple flow records, due to timeouts from lulls in the conversation, the flow table filling up, or the 30 minute flow maximum lifetime. This means that one often has to string multiple flow records together to get all of the data corresponding to an entire TCP session. In these cases, the TCP flags field can be used to determine whether a flow represents data from the start, middle or end of the TCP session. Flows from the start of a session will have the SYN (but not FIN or RST) bit set, flows from the middle of the session will typically have no flag bits set, and flows from the end of the session will have the FIN or RST bits set (but not SYN).

Flows for UDP and ICMP traffic behave similarly, although it is important to note that since neither of these are connection oriented protocols flows of UDP and ICMP traffic are just collections of similar packets.

Terminology

A **NetFlow device** is a Cisco router or switch that supports NetFlow services and which is exporting NetFlow records. **NetFlow Protocol Data Units** (PDUs, also called **NetFlow records**) are the accounting records that NetFlow devices emit. We will use the term **flow records** or **flow logs** to refer to flow accounting records in the internal flow-tools format. A **NetFlow collector** is a host running flow-capture to create a flow log from NetFlow records exported from one or more NetFlow devices. Note that **flows** are unidirectional collections of similar packets. We will use the terms **connection** or **session** to refer to all of the packets associated with bidirectional communication between a client and a server. A connection (such as a connection from a telnet client to a telnet server) will consist of at least two and possibly many more unidirectional flows.

The OSU Flow Tools

The Ohio State University has written a suite of tools for collecting, filtering, printing and analyzing Cisco flows. The tools are written to work as UNIX pipelined commands, making it easy to perform data

reduction without creating unnecessary intermediate files. The tools are grouped roughly as “capture tools,” “general analysis tools,” and “security tools” in the following discussion.

Work on flow-tools started in August of 1996 when Cisco had released an EFT image with a new feature called NetFlow switching. At this time OSU had Internet connectivity via CICNet, and a local peering with the state network, OARnet. We needed a way to determine how much of our traffic was staying local to CICNet, and indirectly the other big 10 schools, and how much was transiting CICNet to MCI. Where our traffic was terminating and originating would potentially influence future Internet bandwidth purchasing decisions. An initial release of flow-tools aided by testing and feedback from other CICNet member schools was generating statistics in early September. Since then features, fixes and documentation have been added to the tool set, primarily for OSU’s internal use with incident response and traffic analysis. Cisco has improved NetFlow switching over the past few years including features such Border Gateway Protocol (BGP) AS information, aggregated flow exports, and integration with dCEF to provide what we are now using, NetFlow accounting. Much of the work OSU has done has been made publicly available in open source form.

Internal Operation

NetFlow records are exported from Cisco gear in one of several versions. To accommodate the slightly different records, the flow-tools package receives these records in native Cisco format and translates the records to a fixed size record stream format which contains a snapshot of what was available in the Cisco version 5 NetFlow export records. This conversion work is done in the programs that receive flow exports from devices (flow-capture and flow-receive), and most of the rest of the tools work with this internal flow-tools representation (flow-fanout is the exception – this is a generic UDP packet multiplexer). Most of the programs in the flow-tools suite were designed to read and write to stdin and stdout, although some have command line options that allow you to redirect I/O to specific files. The tools also support zlib compression (RFC 1950 [10]) on the fly to conserve space.

Capture Tools

Each NetFlow enabled router or switch has to be configured to export their flow records to a flow collector. Flows are exported through UDP packets sent to a designated host and port, where some sort of network service is expected to receive the data and do something useful with it. The IOS command `ip flow-export destination 10.0.0.1 12345` would cause a device to export NetFlow records to the host with IP address 10.0.0.1 at UDP port 12345. Several packages are available for receiving and processing these NetFlow exports, see the related work section at the end of this paper for a brief survey.

In the case of the flow-tools package this collection service is provided through a program called **flow-capture**. Flow-capture listens on the designated UDP port and writes the received flow records to log files. To keep individual logs from growing to an unwieldy size flow-capture rotates to a new file periodically. Flow-capture was designed to facilitate the long-term archival of flow records and has a built-in mechanism to manage the log file space in the output directory, either by restricting the number of separate log files that are maintained or by restricting the total bytes allocated to log files in the output directory. The UDP port, output directory, rotation period, and log count or size limits are all configured through the command line. For instance, `flow-capture -E1G -n23 -p9991 -w/var/log/flows -z6` would cause flow-capture to listen on UDP port 9991, write its logs with level 6 compression to `/var/log/flows`, rotate the file once per hour, and save up to one gigabyte of log files in the output directory.

Flow-capture also allows you to connect to it via TCP to receive a real-time feed of flow records, which you can receive using the `nc` (netcat [8]) program, as in `nc capturehost 9991 | flow-print`. This can be useful for debugging as well as for applications that require real-time access to NetFlow records such as `flow-dscan`.

There is also a simple program called **flow-receive** which listens for NetFlow records on a UDP

port (like flow-capture), but which writes the resulting flow records to stdout rather than archiving them to files. This is useful for debugging NetFlow exports.

Flow-fanout captures NetFlow records through a UDP port (like flow-capture) and replicates them to multiple destinations, which are specified by host and UDP port on the command line. This is primarily useful for debugging purposes, though it can also be used to replicate flow records to other tools, such as `cflowd`.

Flow-mirror and **flow-rsync** are simple scripts that copy flow logs from the collection hosts to the archive hosts using FTP and rsync, respectively.

Flow-expire implements the same space management features that flow-capture does. This is useful for managing the logs on systems that have copies of the flow logs mirrored through the flow-mirror or flow-rsync scripts but which are not running flow-capture.

Our architecture for flow collecting and processing has grown from a single Sparc 5 equipped with a few Gigabytes of disk space in August of 1996 to eleven Pentium and Pentium II based flow collectors, a dedicated Pentium III file server equipped with two 250 gigabyte RAID5 arrays, and two one gigahertz Athlon processing servers. The flow collectors are typically connected directly to the routers they are

Example Topology with Netflow enabled routers

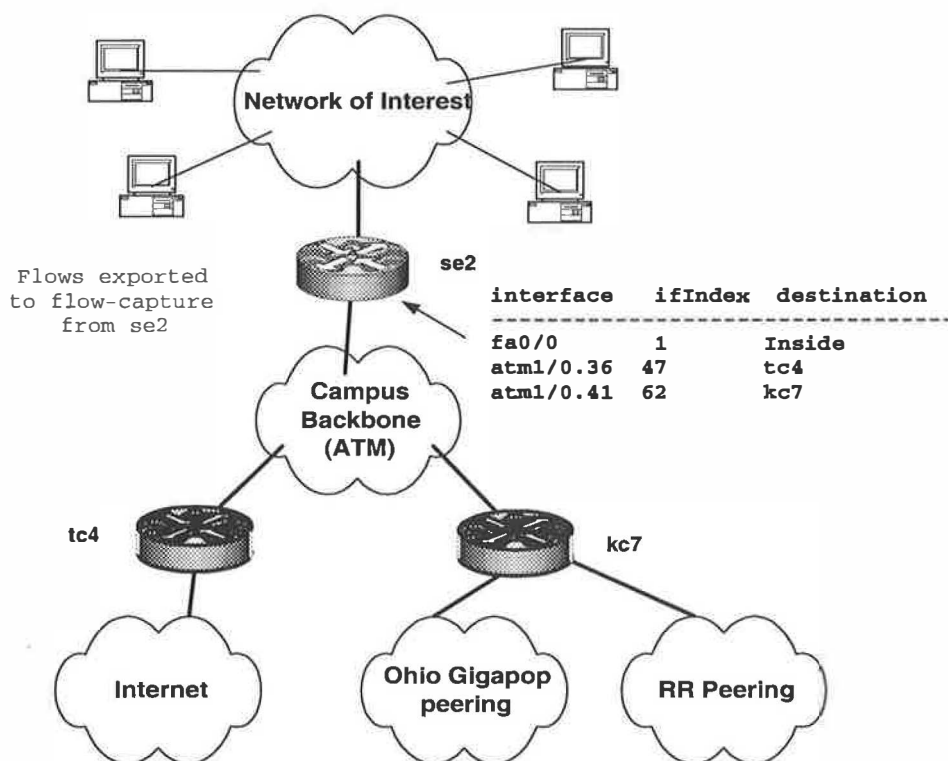


Figure 2: Simplified diagram of the OSU network.

gathering flows from and also can double as reverse terminal servers and consoles for other physically adjacent equipment. These collectors have minimal local disk space, so our primary file server polls the flow collectors once an hour with rsync for completed flow exports. We use flow-capture to receive flows and manage the disk space on the collectors, and flow-expire to manage space on the file server. Figure 2 shows a logical diagram of the OSU backbone and its connections to the Internet. Figure 3 gives an idea of how we manage flow captures.

Flow collector performance depends directly on the availability of enough free CPU cycles to compress the inbound flow exports and move the compressed stream to disk. To verify that the collector is not over subscribed, poll the kernel statistics for dropped UDP datagrams due to full socket buffers. On FreeBSD this can be done with `netstat -s`. Flow-capture attempts to use larger than default socket buffers to help ensure bursty flow exports are handled without unnecessary packet loss. Flow-capture is started with `rtprio` which gives it an unfair advantage over other running processes.

Disk IO and CPU both contribute to the performance of processing the flow data. Initially we would collect and process flows from the campus border on a single server. At that time most of our other routers

were not capable of generating flow statistics due to hardware constraints or were not able to run the Cisco images that supported the NetFlow feature set. Today, NetFlow exports are archived from all the campus backbone routers to a single large disk store which is shared with two other servers dedicated to processing the data sets and generating reports. Each of the servers where the data is crunched also have a striped two disk temporary storage area to reduce NFS traffic when iterating over the same data set many times, as we typically do. A simple benchmark of using flow-cat to uncompress a days worth of flow exports from a single router and print it in ASCII leads to a processing speed of about 300,000 flows per second on the one gigahertz Athlon boxes using local storage. Typical pipelines such as `flow-cat <dataset> | flow-filter | flow-stat <options>` can reduce this down to around 245,000 flows per second, which is adequate for the number of reports we run and the size of the data sets currently in use. Our upgrade plans include a faster RAID controller and a newer version of FreeBSD with improved NFS and network performance.

We currently average 67,320 octets per flow and 92 packets per flow. One of our busiest routers handles 397 gigabytes of traffic per day (about 35 megabits per second) in 548 million packets which generates 5.9 million flow records per day. The flow-

Distributed Netflow collection and processing

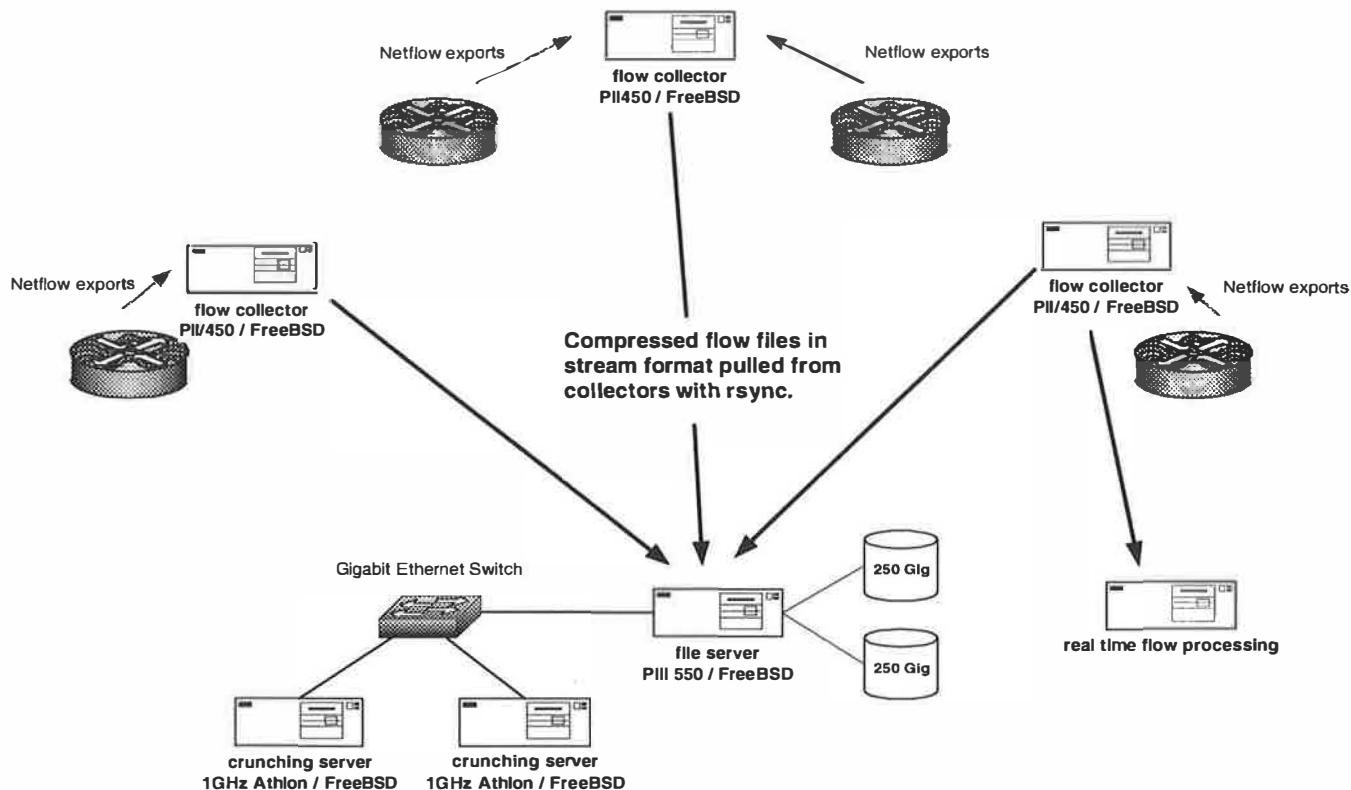


Figure 3: Simplified diagram of flow collectors at OSU.

tools flow record is 60 bytes long, so this works out to about 350 megabytes of flow logs per day at level 0 compression. We use level 6 compression by default, where we get a 4.3:1 compression ratio, so this actually works out to about 82 megabytes of actual disk space. This number can vary considerably – various denial of service attacks (especially SYN floods) can greatly increase the number of flows created, and accordingly, the number of flow records reported.

General Analysis Tools

Since our logs are split into relatively small files that represent network traffic for short time periods, we need a way to catenate these files together prior to processing. We can not simply use the UNIX `cat` program since each log starts with a general header that describes the contents of that log and this header needs to be removed from within the catenated files. The **flow-cat** program reads one or more flow logs (listed on the command line, though it will also read from stdin) and catenates the contents of these files in turn to stdout (or a designated output file).

Flow logs are written as binary records to a file and are not directly readable. The **flow-print** program reads records from a flow log and prints their contents in one of several output formats. For example, `flow-cat * | flow-print -f 5` would print information about each of the flows in all of the logs in the current directory using format 5. Format 5 includes the start and end time of the flow, source and destination IP address and TCP or UDP ports, IP protocol type, source and destination interface numbers, TCP flags, and a count of the number of octets and packets for each flow. In the example in Figure 4 we have removed several of the output fields to make it more readable. The column labeled “p” is the IP protocol type – 6 is TCP, 17 is UDP. The column labeled “f” is the OR of the TCP flags for each flow. The last two columns, labeled “#” and “octets” show the total number of packets and octets for each flow. We also shortened the timestamp format in the “start time” column – normally timestamps are printed as MMDD.HH:MM:SS.SSS, so a timestamp of 0927.18:30:23.562 would represent the time 18:30:23.562 on September 27.

You can use command line options to cause `flow-print` to translate IP addresses and port numbers to names, where possible. We do not use this option

very often, since we are used to working with numeric IP addresses and port numbers. Also, the translation to host names can lead to misleading interpretations due to DNS spoofing (through cache poisoning or other techniques) and due to port overloading (running a web daemon on TCP port 23, which would be reported as a telnet connection).

You can search flow logs and pull out interesting records with **flow-filter**. Flow-filter allows you to match records by the following fields:

- Source or destination autonomous system number (-a and -A options).
- Source or destination port number (-p and -P options).
- IP protocol type (-r option).
- Source or destination IP addresses, using Cisco standard Access Control Lists (ACLs, -S and -D options).
- Input or output interface numbers (-i and -I options).

Most of these options allow you to specify ranges and lists of values to match. For instance, `flow-filter -r 6 -P 21,23,25,80` would match TCP flows (IP protocol type 6) with a destination port of 21, 23, 25 or 80 (which are usually FTP, telnet, SMTP and web services). The ability to filter flow records using Cisco standard ACLs allows us to perform powerful searches through our archives as part of incident investigations. The command `flow-filter -f flow.acl -S attackers -D victims` would read ACL definitions from a file named `flow.acl` (see Figure 5) and match flow records where the source IP address matches the “attackers” ACL and the destination IP address matches the “victims” ACL. You can also use the command line options to further filter the output by other fields, such as by IP protocol or TCP/UDP port.

We wrote a related script named `flow-search` to facilitate investigations of computer intrusions using the flow logs. Flow-search allows you to easily apply the same filter to a large number of separate log files. You could easily do this by applying `flow-filter` to the output of `flow-cat`, but this procedure disassociates the flow records from the name of the log file that they were found in, which can be inconvenient in incident investigations. Flow-search applies `flow-filter` to each log file individually and keeps the results in derivative files named after the source logs.

start time	src ip	src port	dst ip	dst port	p	f	#	octets
00:00:11.380	164.107.1.2	1026	205.188.254.195	4000	17	0	1	56
00:00:11.384	216.65.138.227	1055	164.107.1.3	28001	17	0	1	36
00:00:11.384	164.107.1.3	28001	216.65.138.227	1055	17	0	1	68
00:00:11.392	164.107.1.4	27015	24.93.115.123	1493	17	0	3	1129
00:00:11.392	164.107.1.5	1034	205.188.254.207	4000	17	0	1	48
00:00:11.392	128.146.1.7	53	206.152.182.1	53	17	0	1	61
00:00:11.404	204.202.129.230	80	140.254.1.6	1201	6	3	30	14719

Figure 4: Sample output from `flow-print`, edited to make it more compact.

Since flow records are created when their corresponding flows end the records are recorded in the log file in order of the ending time of these flows. This makes it hard to correctly interpret the contents of the flow logs. For example, you may suspect that an intruder is logging into a host through a backdoor of some sort, and you can see network activity coming from the host (scans, exploit attempts, IRC connections, etc) that lead you to believe that the intruder is active. But the flows associated with the backdoor connection may not show up in the flow logs until 30 minutes after the backdoor traffic actually started, so the other activity may actually occur first in the log. We wrote **flow-sort** to quickly sort the flow logs into chronological order according to the starting time of each flow. Flow-sort is implemented as a rolling heap sort, where flows are sorted into a heap which represents a 35 minute wide window (the maximum duration of a flow is 30 minutes). As flows move out of the rolling 35 minute window they are written out, now sorted by the starting time of the flow. Flow-sort can also save the heap when it reaches the end of the input and can restore from a saved heap when it starts up on a new file, which allows us to correctly sort flows between different log files.

Since flows are unidirectional and do not contain any indication about who initiated a connection, it can be difficult to correctly determine the client/server relationship between the source and destination hosts in each flow. One can apply heuristics based on the transport level source and destination port numbers, or by maintaining state about previous network activity on each host (“host A created a connection to TCP port 21 on host B”) and using that to make inferences about unknown traffic (“this traffic from {TCP, A, 12345} to {TCP, B, 32145} might be a passive mode FTP data connection”). These are subject to mislabeling (“this specific traffic was a backdoor to host B”) or ambiguity (is traffic between TCP port 2000 and port 6000 is either OpenWindows, X, or something else entirely). The TCP flags field is of no help, since the same flags are usually seen in aggregates of packets in both directions. It would be useful if flows of TCP traffic where a TCP packet contained a SYN but not an ACK were marked in some fashion, but NetFlow services does not provide that information.

```
! permit anything
ip access-list standard all permit any

! match the attackers
ip access-list standard attackers permit 10.0.0.1 0.0.0.0
ip access-list standard attackers permit 128.146.222.0 0.0.0.255
ip access-list standard attackers deny any

! match the victims
ip access-list standard victims permit 140.254.1.1 0.0.0.0
ip access-list standard victims permit 140.254.1.2 0.0.0.0
ip access-list standard victims deny any
```

Figure 5: Example access control list definition file for flow-filter.

We can try to use the starting times of the flows to infer the client/server status of the endpoints – the source of the first flow is the one that initiated the connection. Flow-connect attempts to make these inferences from sorted flow records and aggregates flow records for the same session into a single flow record whose source IP is the client and whose destination IP is the server. At the time of this writing, this is still something of a work in progress, though preliminary results are encouraging.

Network Planning and Performance Tools

We have depended primarily on other tools such as statscout [13], MRTG [14] and a variety of home-grown tools for network planning, performance analysis and monitoring. We have found it useful to generate some common reports from the flow logs. **Flow-stat** summarizes flows into useful reports, some of which are easily viewable as graphs using tools like gnuplot. Flow-stat currently supports almost 20 different report styles, including summaries by source or destination AS number, port, or IP address, and source/destination matrix summaries. See the section on network planning and performance tools for some examples of how we use flow-stat.

Flow-profile is similar to flow-stat but provides the ability to aggregate flows based on groups of hosts as defined in a configuration file. This provides a powerful mechanism for additional data analysis, and is especially useful to generate usage data by group which can be used as a source for usage based billing. Flow-profile reads a configuration file which essentially describes the mapping between billing units and IP address ranges. The data from the flow records passed to flow-profile are aggregated according to these groupings and are summarized in the output, which is designed to serve as input to a billing system. See Figure 6 for an example of a configuration file, and Figure 7 for sample output.

The flow-tools package is a small set of programs for processing NetFlow exports that can be chained together, along with other standard utilities such as awk, grep, perl, sort, etc to produce reports. Leveraging off existing well-known utilities can lead to simple one-liners that produce detailed network activity reports or data sets for depicting long term

usage with utilities such as gnuplot or mtv. The following are examples of some of the types of reports that can be generated.

Top 10 Users

It is easy to create custom reports. Here is how we find the list of the top 10 network bandwidth users in our dorm network, ResNet (see Figure 8). We use flow-cat to concatenate the data sets for a day's worth of logs together and pass the resulting output to flow-filter to isolate traffic with an input interface of 1 and an output interface of 47 or 62 (our connections to the outside world, see Figure 2). The output from flow-filter is passed to flow-stat, which we will instruct to

generate a usage report based on source IP address (-f9) and to report the totals in percent/total form. Finally, we filter out the comments with grep and sort the results in descending order by the value in the total octets column. If you examine the numbers you will see that the top 10 users are using almost 30% of the total octets counted.

Counting Addresses

First, let's count the number of unique IP addresses on the Internet that have exchanged traffic with hosts at ResNet. We will use flow-filter to pull out just traffic going to the Internet, and flow-stat format 8 (statistics by destination IP address) to generate

```
# define the inside interface
inside 1 3 4

# define outside interface
outside 8

range 128.146.070.001 128.146.070.001 math
range 128.146.024.001 128.146.024.001 physics
range 128.146.225.001 128.146.225.001 economics
range 128.146.216.001 128.146.216.001 math
range 128.146.222.001 128.146.222.001 athletics
...
range 164.107.005.151 164.107.005.244 physics
range 128.146.050.240 128.146.050.244 psychobotany

# define these after ranges to make sure all addresses in a block
# have range definitions
bound 128.146.0.0 128.146.255.255
```

Figure 6: Sample configuration file for flow-profile.

#group	octets	packets
#	in	out
math	358627478	3423321 3274516064
physics	7449413327	38512040 32410612445
psychobotany	2025644653	27020722 14947199604
english	212751	2283 2108750
...		

Figure 7: Sample output from flow-profile.

```
flow-cat cf05.2000-09-26.* | flow-filter -i1 -I47,62 | flow-stat -P -f9 \
                             | grep -v '^#' | sort -n -r +2 -3
```

IP	flows	octets	packets	duration
164.107.70.189	0.077	7.477	3.669	1.053
140.254.229.202	0.091	5.036	2.487	0.719
140.254.233.119	0.116	3.874	1.936	0.824
164.107.67.66	0.262	2.412	1.240	0.530
140.254.106.226	0.122	2.358	1.306	0.754
140.254.229.255	0.045	1.977	0.969	0.264
164.107.86.143	0.059	1.835	0.988	0.337
164.107.93.206	0.091	1.602	0.857	0.390
164.107.90.220	0.238	1.390	0.742	0.352
140.254.236.103	0.013	1.381	0.714	0.152

Figure 8: Using flow-stat to find the top 10 users.

the unique list, which we will count with the wc (Word Count) program: `flow-cat cf05.2000-09-26.* | flow-filter -i47,62 | flow-stat -f8 | grep -v ' #' | wc -l`.

From this, we see that there were 945,189 unique destination IP addresses contacted.

Similarly, this will count unique source IP addresses at ResNet that have sent traffic to the Internet: `flow-cat cf05.2000-09-26.* | flow-filter -i1 -i47,62 | flow-stat -f9 | grep -v ' #' | wc -l`.

This reports that there are 6,209 unique source IP addresses active.

It might be interesting to count the unique IP addresses at ResNet that have received traffic from the Internet. We will turn the filter around (the destination filter is ResNet's internal network and the source filter is the Internet links) and count the unique destination addresses: `flow-cat cf05.2000-09-26.* | flow-filter -i1 -i47,62 | flow-stat -f8 | grep -v ' #' | wc -l`.

This reports 11,739, which is greater than the actual number of hosts that generated traffic. This implies that the network had been host scanned at least once.

Finally, here is a way to count the Internet hosts that have sent traffic to ResNet:

```
flow-cat cf05.2000-09-26.* | \
  flow-filter -i47,62 | \
  flow-stat -f9 | \
  grep -v ' #' | wc -l
```

This reports 879,438, which is less than the number of hosts that we sent traffic to (945,189). This implies that ResNet hosts were involved in host scanning the Internet.

Security Tools

We most commonly use the flow logs in our various security functions, for incident response,

intrusion detection, firewall planning, and security assessments and consultation.

Intrusion Detection

NetFlow logs are not useful for any form of content based intrusion detection since the flow records do not contain the data portion of the network traffic. We also do not have a detailed picture of the values in all of the headers for individual packets, and so we cannot use that to detect signatures of intrusions that leave calling signs there. We can use the flow logs to detect network access policy violations, to report on the activities of known suspicious hosts, and to detect some of the more obvious forms of scanning and denial of service attacks.

Flow-dscan is an attempt at detecting and reporting interesting network related events in near real-time. It can be configured to report:

- Excessive octets or packets per flow – typically floods.
- A source IP contacting more than a threshold of destinations – host scanning.
- A source IP contacting more than a threshold of destination ports on a single host. The port list is limited to 0..1023 to keep the memory overhead low.

Flow-dscan must keep a fairly large hash table of {source, destination} pairs in memory to be able to detect slow port and host scanning. Memory is allocated and managed dynamically based on the frequency of flow arrival. The age and various table sizes can be user configured. Optional source and destination suppress lists are also supported in hash format for fast lookups. Typically on-line multi-player game servers and web-based add servers must be entered in the suppress list to prevent them from being reported as scans. Flow-dscan can either be run on archived data sets or by connecting to flow-capture for

```
>flow-cat /netflow/se2/cf05.2000-09-28.* | flow-filter -r6 -i1 -I47,62 \
| flow-dscan -b -p -O -P

info(6):  port scan: src=140.254.103.62 dst=24.160.184.73 start=966204748

>flow-cat /netflow/se2/cf05.2000-09-28.* | \
  flow-filter -r6 -i1 -I47,62 -f flow.acl -S portscan | flow-print
```

Sif	SrcIPaddress	Dif	DstIPaddress	Pr	SrcP	DstP	Pkts	Octets

0001	140.254.103.62	003e	24.160.184.32	06	f78	15	1	60
0001	140.254.103.62	003e	24.160.184.32	06	f79	7	1	60
0001	140.254.103.62	003e	24.160.184.32	06	f7a	5d	1	60
0001	140.254.103.62	003e	24.160.184.32	06	f7b	13	1	60

0001	140.254.103.62	003e	24.160.184.32	06	e4b	13	1	60
0001	140.254.103.62	003e	24.160.184.32	06	e4c	5d	1	60
0001	140.254.103.62	003e	24.160.184.32	06	e4d	7	1	60
0001	140.254.103.62	003e	24.160.184.32	06	e4e	15	1	60

Figure 9: Results of running flow-dscan to detect a scan and flow-print to view the associated traffic.

real-time data. Flow-dscan is usually run in the background against a live feed of flow records, and reports its results through syslog. You can also force flow-dscan to run in the foreground and report to stdout, which is useful for interactive use against archived log files (see the section on intrusion detection). Visually inspecting the flows with flow-filter and flow-print can verify the scanning activity.

Normally flow-dscan would be run against a live feed of flow records, but you can also run it against archived logs. In Figure 9 we use flow-filter to pull out TCP traffic from OSU going to the Internet, and then run flow-dscan in the foreground on the resulting flow records. After running flow-dscan we used flow-filter to pull out traffic coming from the source of the scan and print it.

Although it is not practical to respond to every trigger it has been useful to log such activities so that we can refer back to them at a later date. The frequency of external sources scanning our network is so high that an insecure machine will almost certainly end up on someones database of vulnerable hosts within 24 hours of installation. Insecure hosts are frequently compromised within a day of being set up on the network.

A simple script named **flow-scan-report** takes a time range and IP address as arguments and uses flow-filter and flow-print to display flow activity to and from that address in the given time range. This makes it easy to pre-compute brief snapshots of network activity for each of the detects that turn up in IDB (our Incident Database, in the section "Related Work").

Incident Response

As we saw with the previous example, flow-filter is an effective tool for pulling interesting traffic out of the haystack. This is invaluable for incident response. For example, if we receive a report that one of our computers was involved in a scanning and intrusion incident at another Internet site, we can use the flow logs to:

- Confirm whether the alleged incident actually involved OSU.
- If it did, we can usually use the flow logs to determine what hosts the OSU host contacted by using flow-filter to search for traffic coming from the OSU host.
- We can also search for traffic going to the OSU host to determine whether it is being controlled from elsewhere.
- Once we identify the hosts used to compromise our hosts, we can search the flow logs for traffic from those hosts to OSU to discover other hosts that might have been compromised.

Iterating over the flow logs with varying options to flow-filter, flow-stat, and flow-print on each pass allows us to quickly determine to source(s) and destination(s) of DoS attacks and potentially the attacker and their arsenal of compromised hosts. Once the

compromised hosts, victims, or attackers are isolated the IP addresses can be quickly disabled by use of a black-hole router which injects specially tagged prefixes into our backbone routers which get rewritten using route-maps to point to a non existant destination.

For example, early in the afternoon of July 3, 1999 we were alerted of slow or non-responsive network services on campus. It didn't take long to find that the inbound side of our OC3 connection to OAR-net was full. Running the most recent flow logs through flow-filter to isolate inbound traffic and flow-stat to create a summarized traffic report by destination IP isolated the destination to a single host. A second run of the flow logs through flow-filter to isolate flows to that single IP revealed thousands of ICMP echo replies – a Smurf attack. Further analysis of the destination IP flow logs revealed an IRC client session which is a common ingredient on provoking a denial of service attack. Disabling the host with a black-hole route prompted the attackers to end the attack end shortly after.

Unfortunately the Smurf attack was only a precursor to the activity that followed later that day. Shortly after one of the evening fireworks displays our upstream provider sent a page informing us that severe denial of service attacks originating at OSU required them to shut down OSU's Internet connection. Most departments at OSU at this time were connected with 10 megabit per second Ethernet to the campus backbone, so in order to generate a full OC3 of traffic we knew that many hosts on many different LAN segments must be involved. Again, flow-filter was used to isolate outgoing traffic by filtering on the interface fields and flow-stat was used to generate a report based on destination IP, and we discovered the IP address of a single victim. We soon discovered that many hosts on the OSU network were sending high bandwidth UDP streams to the victim. Further analysis of a larger window of the flow logs using flow-filter and flow-stat to create a report of source IP addresses contacting the victim revealed about 43 sources on many different LAN segments participating in the attack. Using those sources in a filter we also found that multiple victims were involved over the course of the day.

Many of the compromised hosts on campus were not very active on the Internet, so it was easy to spot the IP address of the attacker in the flow logs and the TCP port used to start the UDP floods. The attack was controlled by a fairly simple set of Perl and shell scripts that connected to compromised hosts through a shell backdoor on the FTP port, from which it would run a script to download a UDP flooder called milk from another university through the Remote Copy Protocol (RCP). The milk script would then be run against one of several external targets.

Using black-hole routes to disable the compromised hosts, victims, and attacker proved effective in

stopping the attacks. Later long term analysis of the flow logs for the compromised hosts revealed the method of break-in and several sets of hosts that were involved in what was apparently a distributed and automated scan and exploit. One last iteration over the archived flow records after the incident revealed that as many as 250 hosts at OSU were compromised in the initial set of breakins on July 2, although only about 50 of them were used for the UDP denial of service attack the next day.

The use of flow logs to home in on compromised hosts and their traffic has shown that it is not uncommon for a site to be scanned for vulnerabilities by one host, compromised at a later date by a second, contact a third site for downloading of denial of service and exploit tools, and then have the installed and waiting remote controlled denial of service programs be triggered at a later date by yet a fourth site to attack a victim.

Firewall Planning and Security Assessments

As we have instrumented the rest of our core routers and switches with NetFlow exports, we have come to increasingly rely on the flow logs for guidance in designing firewalls and in studying the network behavior of systems in site and product security evaluations.

Host Activity Profiling

Flow-host-profile builds a list of the network services running on each host on campus, and allows us to compare activity over a period of time with the existing profile to detect changes. We are especially interested in new hosts and new services that show up on campus. The sample output from flow-host-profile (Figure 10 shows activity for several services that we had not previously seen (e.g., HTTP services on 128.146.1.4). The new activity on port 7440 on 128.146.1.3 might be a new service, but sometimes busy clients have high end port numbers that show up in the report (as they get reused the connection count goes up, passing our activity threshold). We've been experimenting with looking at not just the presence of new services (and hosts) but also with changes in the level of activity of a service. For example, it would be interesting to know if activity on a usually quiet FTP server suddenly increases (possibly due to warez trading through a writable directory, for example). Unfortunately, flow-host-profile is very sensitive to traffic that evades its attempts to winkle out the client/server role of each endpoint, and tends to either create many false positives or becomes insensitive to low levels of

intrusion activity if we turn the thresholds up to decrease the false positives.

Common Problems in Interpreting NetFlow Logs

We already discussed the difficulty of determining the client/server role of the hosts at each end of a flow (in the section on general analysis tools) and of interpreting the flows in light of the fact that they are ordered chronologically by the ending time of each flow. There are a number of other issues that you need to keep in mind when you are reading through flow logs.

It is important to recognize that source IP addresses are easily spoofed. This is common in many denial of service attacks or in chaff created by scanning tools like nmap. If you have unicast Reverse-Path Forwarding (RPF) checking enabled on your devices, or use Access Control Lists (ACLs) to drop spoofed traffic this traffic will be recorded in your flow exports with a destination interface of 0 (indicating that the traffic arrived, but was not forwarded). Of course, hosts within a LAN can still spoof the addresses of other hosts on the same LAN.

Note that NetFlow enabled devices only create flow entries for the traffic that actually passes through that device. If you have asymmetric routing conditions where outbound traffic passes through a different router than the analogous inbound traffic and you are only looking at the flows from one router, you will only see part of the flows representing those client/server sessions. To get a complete picture of the traffic in these cases you would have to combine the flow records from the devices that handle all of the traffic (through flow-cat and flow-sort). Note that since flow-sort uses a 35 minute window for sorting, you need to merge the files in chunks that span less than 35 minutes. This presumes that the clocks have been accurately synchronized to a common time source.

One issue that needs to be considered when using flow exports from the routers is the reliability of the data sets. Flows are currently exported via UDP with no ability for the collector to signal retransmission if it detects missing NetFlow PDUs. Each PDU contains a 32 bit sequence number that can be used to detect missing or out of order flows. Out of order flows could be an indication of an attacker trying to inject false PDUs into the collector. Missing flows could either indicate the flow-collector is overloaded, possibly due to an attack or the network segment

IP-ADDRESS	PORT	SERVICE	PROTO	CONNECTIONS	DAYS	PERCENT
128.146.1.1	53	domain	17	3	1	100
128.146.1.2	162	snmptrap	17	3	1	3
128.146.1.3	7440	N/A	6	3	1	100
128.146.1.4	80	http	6	3	1	100
128.146.1.5	518	ntalk	17	3	1	0

Figure 10: Abbreviated output from flow-host-profile, showing new hosts and services.

connecting the flow collector to the router is over subscribed or under attack. The possibility of spoofed flows can be minimized by deploying unicast RPF or IP spoofing filters appropriately. Overloaded network segments can be avoided by directly connecting the collectors to a dedicated router interface on the router exporting the flows and limiting traffic on that network with access lists. Cryptographic signatures on flow PDUs and a reliable transport mechanism could reduce some of the potential problems, but not without the memory and CPU penalties on the routers and line cards. The potentially unreliable export mechanism and 32 bit sequence number is adequate for our current needs.

Privacy and Legal Concerns

The flow logs do not contain a record of what is usually considered the contents of the packets. This means that although we could determine that a given host accessed a given web server at a certain time, the flow logs would not contain a record of the URL requested or the response received. However, if you can correlate the activity recorded in the flow logs against the data in other logs (such as authentication logs), you might be able to match accounts (and so, to a large degree, people) to IP addresses, IP addresses to their associated network activity, and then match that network activity to specific details such as URLs requested, email addresses for correspondents, newsgroups read and so on. Consequently, the act of recording and archiving NetFlow records raises a number of privacy concerns.

In addition, OSU is a state university and as such is subject to the state public records laws, which indicate that most records created as part of the normal business processes of the university are "fair game" for disclosure requests from the general public. This of course raises a fair degree of concern that someone might request a copy of our NetFlow logs to determine what our employees are using their computers for. On the other hand, we are also subject to FERPA (Family Education Rights and Privacy Act) which indicates that student academic records are exempt from the public records laws. According to our lawyers, it is not clear whether the flow records for students would be considered protected by FERPA, or whether they are part of the public record. Since the flow logs cover a mixed population of students and non-students, and since we have no easy way to separate them, they enjoy a sort of murky, though dubious legal protection.

We try to protect the privacy of our customer base by storing the logs on secure servers, with limited access by staff members, and with clear access and use policies in place. We do not archive the raw logs to tape, although we do retain a fairly long window (currently about four months worth) on our central file server.

Our rationale is that the logs are invaluable for security, performance and network monitoring and usage based billing. We could aggregate the data and use that for some of these functions, which would solve most of the privacy concerns. However, having a long (2 to 3 month) window of past logs is invaluable for incident response, and we expect that it may prove invaluable for bill dispute resolution as well. We think that the level of detail present in the flow logs represents an acceptable balance between utility and privacy for our environment. On the positive side, we have found that we have had to do content based sniffing (e.g., with tcpdump [15]) far less often, since we have a ready source of information about network activity.

Related Work

Other groups have also been working on tools for collecting Cisco flow logs. In particular, you might be interested in looking at the CAIDA tool collection [16]. In particular, the cflowd [12] system provides a mechanism to collect, save and aggregate NetFlow exports and view a variety of summaries and graphical views of the collected data. There are also a number of tools that can be used with the cflowd package to create reports and graphs.

Cisco has also released a set of tools for collecting NetFlow logs [2]. These tools provide fairly sophisticated data aggregation, graphing and billing features, and as you might expect, support all of the versions of the NetFlow exports, including version 8 aggregated NetFlows.

OSU has integrated intrusions detected through flow-dscan into its Intrusion Database system (IDB). IDB is a web front end to a list of intrusion detects from a variety of sources, including host based scan detectors, snort and now flow-dscan. Our incident response staff reads through the detects in IDB and responds to them in a variety of ways. A hook from IDB allows the operator to easily view a summary of network traffic matching the detect, which is derived from the archived flow logs. Incidents that turn into full-fledged investigations are tracked through our incident tracking system, SITAR [9], which also has hooks that allow us to view the results of previous searches through the flow logs or to initiate new searches.

David Brumley at Stanford University has written a program that converts **argus** [1] logs into flow-tools format, allowing you to use this as an additional source of data for flow-tools [5]. Dave has also written a small perl script for detecting scans through threshold violations [6].

Larry Lidz at the University of Chicago modified the **extract** program from the netlog package [4] package to read the logs that flow-tools creates [7]. The resulting program, **flowextract**, allows you to easily create watch lists to report network activity for

known hostile hosts, or for critical hosts in your environment, or to identify traffic that might indicate hostile activity (for instance, traffic to TCP port 31337). Larry has also written a program called **flow-merge** which merge sorts flow logs from multiple routers into a single log. This is helpful in cases where multiple routing paths lead to asymmetric routing conditions, or where you need to search all of your connections to the Internet for intrusion activity.

Simon Leinen has put together a great summary of flow related references and tools which is well worth looking at [11].

Future Plans

We are often asked why we have not merged our work with the cflowd package. There are several reasons, chief among them being motivation and philosophy. We have not had a need to use the presentation and analysis tools that cflowd provides since we have other tools to do that work. Cflowd appears to have been designed to facilitate the aggregation of data directly from NetFlow exports, and most of the tools written to process flow data from cflowd work with these aggregations rather than with raw flow data. The NetFlow capture mechanism in the flow-tools was designed to efficiently store and manipulate flow records in compressed form, and data aggregation is done by tools operating on these raw logs (if at all). Its design also allows us to take a highly distributed approach to data collection and analysis. It is possible to either write something to export the flow-tools logs into whatever format the cflowd package uses internally, or to duplicate the incoming UDP NetFlow records into parallel incarnations of flow-capture and cflowd.

The principle problems with using the flow logs for intrusion detection are that it is difficult to correctly determine the client/server role of the two endpoints of a flow and that the flows do not contain packet contents. We have not converted our intrusion detection systems to use the sorted flows from flow-sort – we expect that this would allow us to both increase the sensitivity of flow-host-profile and flow-dscan, and also greatly reduce the frequency of false positives.

Two papers by Yin Zhang and Vern Paxson describe their work with detecting backdoors [17] and detecting stepping stones [18] using packet size, packet timing characteristics, and correlations between flows of traffic. Aggregating packets into flows obscures some of the detailed information that they used in their algorithms (individual packet sizes, delays between packets), but it might be possible to adapt their work to flow based data using average packet sizes from the flows and calculated average delays, or using flow sizes and delays between the flows. We hope to start working on this soon.

We will soon be adding support for the NetFlow version 7 PDU (used on some Cisco switches). Cisco

also has a version 8 PDU which is used for exports of aggregated flow data [3]. We plan to add this at some point, but doing so will require development of a different set of analysis tools, since the current tools are designed to work with flows, and not with the aggregated data in tables that are supplied in the version 8 PDU.

We also plan to add support to flow-capture to use the sequence numbers in the version 5 and version 7 NetFlow export headers to detect and report missing NetFlow exports. This addresses some of our concerns about using the flow logs in incident investigations since we will at least be able to tell when critical records are possibly missing, or note that there are no missing records.

The current filtering mechanisms in flow-filter are sufficient for a wide range of tasks, but it would be nice to provide more powerful filters. We plan to fully implement Cisco extended access control lists.

We are also working toward building more powerful interactive front ends for browsing our flow logs.

Conclusion

Cisco NetFlow exports and the flow-tools package contribute to our ever growing toolbox of network management resources at Ohio State University. We are collecting and archiving flow exports from twenty routers with eleven flow collectors, one file server and two high performance data crunching servers. Our campus network has over 500 LAN interfaces, 15 remote sites and multiple external connections including OARnet, Abilene, and a local peering with the Columbus cable modem service provider. An online archive of 500 Gigabytes of compressed flow exports allows turning back the clock on network disruptions and security incidents to provide a post-mortem birds eye view of events leading up to and surrounding an incident. Real-time processing of flow data can shed light on difficult to pinpoint activities by allowing us to view traffic patterns without having to deploy sniffers or LAN probe devices on every LAN segment or WAN link. Departments traffic reports can be generated for potential future cost recovery of network services, and over all network usage reports are used to determine the impacts of popular new applications such as Napster.

The OSU flow tools are available at <http://www.net.ohio-state.edu/software/flow-tools.shtml>. We also have a mailing list (flow-tools@net.ohio-state.edu) – you can subscribe by sending an empty message to flow-tools-subscribe@net.ohio-state.edu.

Mark Fullmer wrote the bulk of the OSU flow tools collection. Steve Romig mostly just uses the tools and makes suggestions for further development, though he has recently been seen writing documentation, fixing various bugs, and finding students to write tools like flow-sort, flow-host-profile and flow-connect. Ron Luman wrote flow-sort and flow-connect.

Suresh Ramachandran wrote the flow-host-profile program.

Biographies

Steve Romig in charge of the Ohio State University Incident Response Team, which provides incident response assistance, training, consulting, and security auditing service for The Ohio State University community. In years past Steve has worked as lead UNIX system administrator at one site with 40,000 users and 12 hosts and another site with 3,000 users and over 500 hosts. You can reach him by phone at 1-614-688-3412 (GMT-0400/0500) or by email at romig@net.ohio-state.edu.

Mark Fullmer is a recovering system and network administrator from a large.edu, currently working on his degree at The Ohio State University and is employed part time in the OARnet engineering group.

Bibliography

- [1] Bullard, Carter <chellyaz@aol.com>, *Audit Record Generation and Utilization System (Argus)*, ftp://ftp.andrew.cmu.edu/pub/argus.
- [2] Cisco, *Cisco Netflow Flowcollector*, http://www.cisco.com/univercd/cc/td/doc/product/rtrmgmt/nfc.
- [3] Cisco, *Netflow services and applications white paper*, http://www.cisco.com/warp/public/cc/pd/iosw/ioft/neflct/tech/napps_wp.htm.
- [4] Schales, Douglas Lee, David R. Safford, and David K. Hess, "The TAMUSecurity Package: An Ongoing Response to Internet Intruders in an Academic Environment," *Proceedings of Fourth USENIX UNIX Security Conference*, anon ftp at coast.cs.purdue.edu in /pub/tools/unix/logutils/netlog, 1993.
- [5] Brumley, David <dbrumley@theorygroup.com>, *Argus Export Scripts*, contact Dave by email for copies.
- [6] Brumley, David <dbrumley@theorygroup.com>, *How to detect network scans*, http://www.theorygroup.com/Theory/scans.html.
- [7] Lidz, E. Larry <ellidz@eridu.uchicago.edu>, *flowextract*, http://security.uchicago.edu/tools/net-forensics.
- [8] Hobbit <hobbit@avian.org>, *netcat*, http://www.l0pht.com/weld/netcat.
- [9] Assor, Mowgli <mowgli@net.ohio-state.edu>, *Security and Incident Tracking And Response (Sitar)*, http://www.net.ohio-state.edu/software/sitar.shtml.
- [10] Gailly, J-L. and P. Deutsch, *Zlib compressed data format specification version 3.3*, http://www.ietf.org/rfc/rfc1950.txt.
- [11] Leinen, Simon <simon@switch.ch>, *Flow based monitoring and analysis*, http://www.switch.ch/tf-tant/floma.
- [12] The CAIDA Web Site, *cflowd: Traffic Flow Analysis Tool*, http://www.caida.org/tools/measurement/cflowd.
- [13] Statscout, *Statscount network performance monitor*, http://www.statscout.com.
- [14] Rand, Dave <dlr@bungi.com> and Tobias Oetiker <oetiker@ee.ethz.ch>, *Mrtg: Multi Router Traffic Grapher*, http://mrtg.hdl.com/mrtg.html.
- [15] Leres, Craig, Van Jacobson, and Steven McCanne, *The tcpdump software package*, anon ftp from coast.cs.purdue.edu in /pub/tools/unix/tcpdump.
- [16] Various, *The Caida Web Site*, http://www.caida.org.
- [17] Zhang, Yin and Vern Paxson, "Detecting Backdoors," *Proceedings of Ninth USENIX Security Symposium*, 2000.
- [18] Zhang, Yin and Vern Paxson, "Detecting stepping stones," *Proceedings of Ninth USENIX Security Symposium*, 2000.

FlowScan: A Network Traffic Flow Reporting and Visualization Tool

Dave Plonka – University of Wisconsin-Madison

ABSTRACT

Internet traffic flow profiling has become a useful technique in the passive measurement and analysis field. The prerequisites for flow-based measurements are now available within the network infrastructure – particularly, in popular Cisco network devices. The integration of this feature has enabled the “flow” concept to become a valuable tool for the network administrator, as it had been in the past for the researcher.

This paper describes FlowScan, a software package for open systems that is freely available under the terms of the GNU General Public License. FlowScan analyzes and reports on flow data exported by Internet Protocol routers. It is an assemblage of perl scripts and modules and is the glue that binds together other freely available components such as a flow collection engine, a high performance database, and a visualization tool. Once assembled, the FlowScan system produces graph images, suitable for use in web pages. These provide a continuous, near real-time view of the network traffic through a network’s border.

Although there are now a number of tools available that collect and process flow data, there is a dearth of visualization tools. By utilizing freely available software tools, FlowScan can be readily deployed in most modern educational institution, corporate, and ISP networks. The information presented by FlowScan assists in understanding the nature of the traffic that your network is carrying. It can be useful in the identification and investigation of anomalies such as poor performance and attacks on hosts. It can provide a foundation on which to develop usage-based billing or to verify the effectiveness of Quality-of-Service policies. By understanding the flows of traffic carried by the network, your institution should be able to make informed network management and bandwidth provisioning decisions.

Introduction

To better understand the nature of Internet traffic, the notion of *flow profiling* was introduced within the networking research community, and subsequently extended by other researchers, producing a series of useful findings. Flow profiling had been predicted to be relevant to applications such as route caching and usage-based accounting [ClaffyPB]. Today, based in part upon market demands for performance and accounting, flow profiling is built into networking devices. While not yet standards-based, the flow methodology is robust enough to persist through this period of vendor-specific implementations, and its benefits in many production networks warrant its early adoption.

Network administrators who collect measurement data often find that they either have collected too little data or too much of it. In a sense, flow profiling is a “sweet spot” between those extremes. Flows strike a balance between detail and summary. They are neither captured packets, nor are they merely aggregate totals tallied as packets travel across a given port or interface. Flows are an expressive abbreviation in which each flow represents a series of packets traveling between “interesting” end points. While flow features within the network infrastructure are a

convenience, the presence of this feature alone is not sufficient for reliable continuous use in production networks. We need software tools to extract, record, and help us understand the flows.

A variety of tools for flow-based measurement have arisen from both the commercial and free software communities [NetFlow], FlowScan is one such freely available system. FlowScan is an assemblage of perl scripts and modules gluing together other freely available components, described below. Similar to other systems aimed at helping users to make sense of an overwhelming quantity of data, FlowScan means to simplify the *collection*, *storage*, and *visualization* of such data. Like most software tools, FlowScan has both ancestors and other relatives that played a part in defining its characteristics.

Several tools have set precedents for collection of network traffic data using passive measurement techniques. Of the freely available tools, MRTG [MRTG, Oetiker] and Cricket [Cricket, Allen] are among the most popular. Typically, these tools collect measurements by periodically collecting SNMP values, such as the interface or port counters named `ifInOctets` and `ifOutUcastPkts`, from routers and switches. They subsequently store this data in a reduced form that is easy to manage since it does not burden the user with database management responsibilities. These

tools are ultimately effective because they make the data available to the end user in a useful, convenient manner. The design goals of FlowScan are similar to those of these tools.

FlowScan analyzes and reports on NetFlow data collected by CAIDA's cflowd [cflowd], a mature flow collection and analysis tool. This flow data is scavenged by FlowScan, which simply tries to discover interesting things about those flows, and maintains counters that reflect what is found. FlowScan then stores this myriad of counter values using Tobi Oetiker's RRDtool [RRDtool]. RRDtool is a database system built from the ground up to effectively store and report on time-series data. Lastly, through the use of RRDtool and other front-ends, FlowScan provides reporting capabilities and visualization of the processed flow data.

FlowScan's Functionality

FlowScan and its component parts are responsible for collecting and processing raw flows exported from routers. FlowScan examines each flow and maintains counters based upon that flow's classification. FlowScan then periodically reports its results and may optionally take other actions. It may be configured to either archive or discard the raw flows after processing.

FlowScan, in its current form, supplies two report modules that illustrate its functionality: CampusIO and SubNetIO. CampusIO is a full-featured report module that is often the first and only report run by most FlowScan users. As such, its features are often thought to be those of FlowScan itself. The CampusIO report interrogates the raw flows, accumulates total counts and pushes these numeric statistics into high performance time-series Round Robin Databases [RRDtool]. Each database contains *packet*, *byte*, and *flow* counters. These counters are maintained in both in and out directions when appropriate.

Each Round Robin Database created by FlowScan contains between one and eight traffic statistics, stored at five-minute intervals, based upon one of these flow attributes:

- the IP protocol such as ICMP, TCP, and UDP
- the well-known service or application such as ftp-data, ftp, smtp, nntp, http, RealMedia, Quake, and Napster
- the class A, B, C network, or CIDR block in which a "local" IP address resides
- the AS (Autonomous System) pair between which the represented traffic was exchanged

Additionally, FlowScan maintains general traffic databases that contain total, multicast, and traffic involving unknown networks.

Figure 1 is a sample FlowScan graph of our campus traffic over a period of 24 hours, demonstrating some of the information handled by CampusIO.

This graph shows several features:

- There is a circadian rhythm to our campus traffic, with the low point centered about 6 AM and with peaks in the late evening.
- There is more total outbound traffic than inbound. That is, our campus consistently provides more Internet content than it consumes, regardless of whether or not our web cache is enabled.
- There is a significant amount of outbound FTP DATA content. This metric is a combination of both ftp-data and PASV mode ftp data transfers.
- Napster users were responsible for an amount of traffic both inbound and outbound that rivals or exceeds both general web traffic (HTTP) and file transfer traffic (FTP) [Plonka].
- Some sort of Napster outage occurred at approximately 3 PM local time.

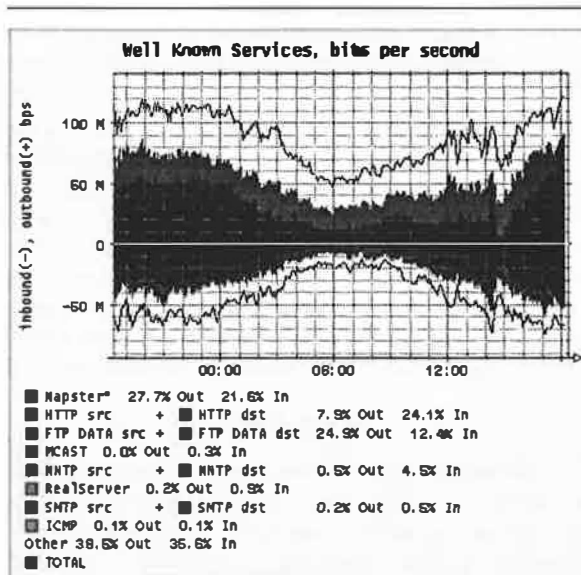


Figure 1: Graph of UW-Madison's campus traffic I/O during 24 hours, Sep 18 & 19, 2000.

The SubNetIO report requires a bit more configuration on the part of the installer, but it subsequently provides all the functionality of CampusIO, plus it maintains per-subnet statistics for applications such as billing a given campus "customer" based on their subnets' usage of precious bandwidth in and out of the campus itself.

Background on Flows and Cisco NetFlow

FlowScan utilizes flows defined and exported by Cisco's NetFlow feature. This flow definition is essentially that which was introduced in [ClaffyPB]. By this definition, an IP *flow* is a unidirectional series of IP packets of a given protocol, traveling between a source and destination, within a certain period of time. The source and destination are defined by IP address. Because the flow is unidirectional, nearly all useful exchanges between two hosts, such as a client and a

server, are represented by at least two flows – one flow in each direction. For TCP and UDP flows, this definition considers the port number to be part of the source and destination address, making it convenient to determine which “well known” application such as HTTP and FTP is likely to have been responsible for the traffic represented by a flow. Additionally, Cisco NetFlow PDUs include the router’s source and destination interface (named by the SNMP “ifIndex”) and NetFlow version 5 PDUs include source and destination netmask and Autonomous System which are of interest to LAN and WAN engineers, respectively. FlowScan-1.003 requires Cisco NetFlow version 5 PDUs and makes use of all of these flow attributes.

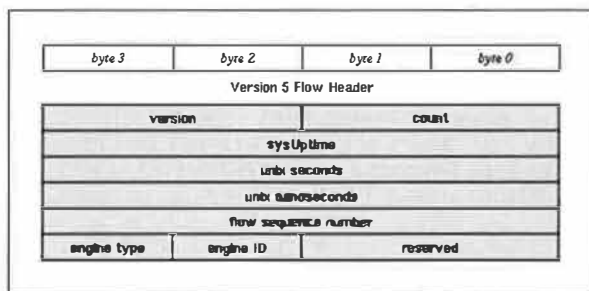


Figure 2: An image of Cisco’s NetFlow V5 PDU header [McRobb].

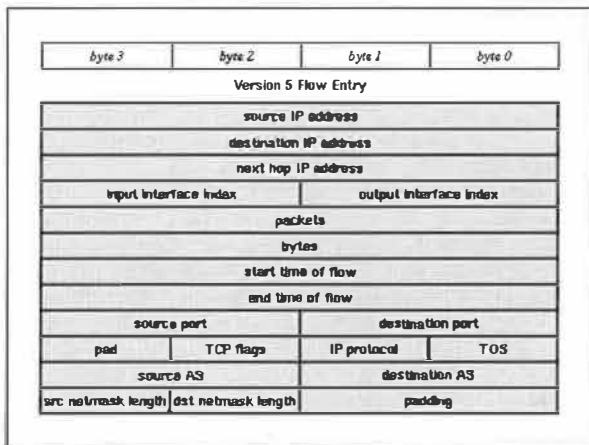


Figure 3: An image of Cisco’s NetFlow V5 PDU entry [McRobb].

Note that while the term “flow” refers to the series of packets itself, NetFlow and FlowScan users often refer to the IP traffic flow accounting record that is exported from the router, and subsequently analyzed, as a “flow”. This usage is natural because it is this exported accounting data that is the tangible data object that FlowScan manipulates. So, in this paper the term “flow” will most often have the latter meaning. Remember that this usage differs from the formal “flow” definition used in some other written works such as [ClaffyPB].

Cisco’s NetFlow version 5 flow export format [Cisco]:

Version 5 flow-export packets contain a flow header followed by a number of flow entries. The number of flow entries in the packet is in the count field in the flow header.

Unlike version 1 flow-export, version 5 flow-export has AS numbers and netmask lengths for the source and destination. [McRobb]

While the NetFlow V5 PDU is well documented in [McRobb] and [Cisco], Figures 2 and 3 are here as a catalog of the flow attributes that are available to FlowScan. This set of attributes enables FlowScan’s current capabilities, and to a degree ultimately limits them by imposing performance requirements and measurement compromises as described in the section on FlowScan problems.

FlowScan’s Architecture

Hardware

A diagram of a basic FlowScan system’s hardware components is shown in Figure 4.

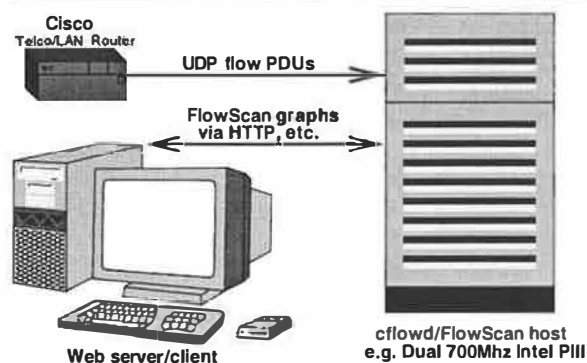


Figure 4: An image showing the basic hardware for the FlowScan System.

While FlowScan does not have strict platform requirements, most users who have successfully deployed it have dedicated either a SPARC machine running Solaris or Intel machine running GNU/Linux or BSD as a FlowScan system. Deploying on one of these platforms is also convenient because cflowd builds and runs on them as well, which allows one to co-locate FlowScan with cflowd so that both have access to the local disk. The fastest FlowScan machines appear to all be multi-processor Intel machines.

For further information, the installation documentation provided in the FlowScan distribution package discusses some recommended hardware parameters such as disk space and network interface card.

Software

A FlowScan system consists of a number of software components. The first such component is cflowd [cflowd], which is described thoroughly in [McRobb2]. FlowScan uses cflowd strictly as a flow

collector. As such, the cflowd components used by FlowScan are the cflowdmux and cflowd programs. cflowdmux receives UDP Cisco version 5 flow PDUs from routers and passes them to cflowd which writes them to disk in a portable, well-defined format of its own. FlowScan requires that the installer apply a patch to the cflowd sources. This modification enables cflowd to rotate and time-stamp its flow files at FlowScan's pre-defined sampling interval, which is typically five minutes. The decision to use five-minute samples was influenced by the popular tool MRTG [MRTG].

The second component is a program called flowscan; note that its name consists of only lower-case characters. "FlowScan" is the package whose primary procedural component is the program flowscan. This program is a perl script that is the central process in the system. It loads and executes report modules of the administrator's choosing. These report modules are simply perl modules that are derived from the FlowScan class defined in the FlowScan.pm perl module. FlowScan reporting modules are described below in the architecture section. As such, it is the flowscan script that actuates the whole system by maintaining databases of statistics regarding the IP traffic represented by the flows.

The third major component of FlowScan is RRDtool. RRDtool is described in [RRDtool] and is well documented in the supplied on-line manual pages. The FlowScan system uses RRDtool to store numerical time-series data and automatically distill or aggregate it into averages over time. Using RRDtool in this manner essentially replaces cflowd's arts++ data aggregation features, which have a different API, and no integrated graphing features such as those built into RRDtool. Specifically, RRDtool is used by FlowScan's supplied report modules to maintain a set of RRD files that form an extensive database of IP flow metrics. Also, RRDtool and RRGrapher are responsible for producing output such as graphs of IP traffic as GIF or PNG format image files.

The other components of the FlowScan package are utilities such as the make command, the Unix cron job-scheduling facility, and the gzip compression utility. Also, there is the flowdumper utility, supplied with the Cflow package, which is used to examine the raw flows "manually".

Figure 5 is a diagram of FlowScan's components and the data objects on which they operate.

FlowScan uses the disk as a large buffer area in which cflowd writes raw flow files that wait to be post-processed by flowscan. This buffering is an important fail-safe when used in networks with very high traffic or flood-based DoS attacks because FlowScan sometimes develops a backlog of flow files yet to be processed, which could total gigabytes in size.

Cisco's NetFlow was chosen as the base technology because of the ease of development within our campus' existing infrastructure.

Together with cflowd patches [patch], FlowScan enables multiple flow log analyses in one pass:

- flow archiving for post mortems
- real-time analysis with cflowd
- near real-time or post-processing with FlowScan

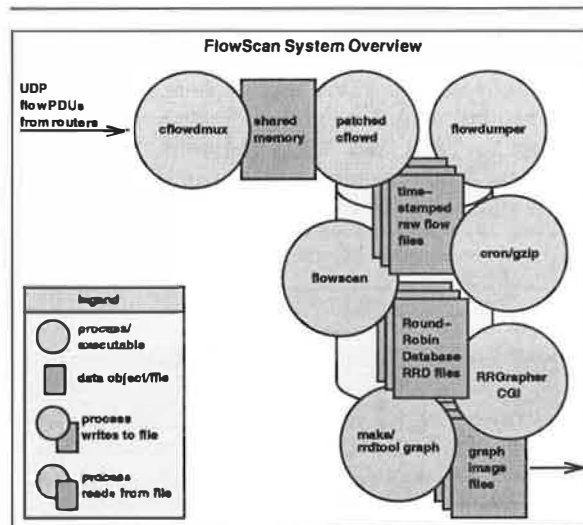


Figure 5: An overview of the software components of the FlowScan System.

Anatomy of a FlowScan Report Module

A FlowScan report module is an object-oriented perl module that has FlowScan as its base class. The module named FlowScan.pm implements the base reporting class, and serves as an example of what a FlowScan report must provide. Objects of a FlowScan-derived class have 3 methods, a.k.a. subroutines. These are:

perfile – Before flowscan processes a given time-stamped raw flow file produced by the patched cflowd, it invokes the perfile subroutine. As such, it is called once *per file*. The name of the file about to be processed is passed as an argument to this routine. For the convenience of derived classes, perfile also converts the time-stamp embedded within the raw flow file's name to a native Unix time_t representation. The names of the raw flow files produced by the patched cflowd are of the "flows.YYYY-MDD_HHMISS+TZ" format. For example, "flows.20000917_20:08:14-0500" is a file created at 8:08 PM on September 17, 2000 in a locale that is five hours west of GMT.

wanted – As flowscan reads each of the raw flows from within a cflowd-produced flow file, it invokes the wanted routine once per flow. This routine decides whether the current flow is *wanted*, i.e., whether it is an interesting flow for the report. The wanted subroutine is at the heart of a FlowScan report because it provides the opportunity for a report to interrogate the values stored in each flow and act accordingly.

report – After flowscan processes the last raw flow in a given flow file, it invokes the `report` routine. This routine is responsible for dispatching with any information “discovered” and collected by the aforementioned `wanted` routine. As such, this routine *reports* what has been discovered about the flows analyzed within the current flow file.

Two report classes are supplied with FlowScan-1.003: “`CampusIO`” and “`SubNetIO`”. `CampusIO` is implemented in the file `CampusIO.pm`, following the usual naming convention for perl modules that implement object-oriented classes. `CampusIO` is derived from the `FlowScan` class defined in `FlowScan.pm`. `SubNetIO` is, likewise, implemented in `SubNetIO.pm` and its base-class is `CampusIO`. That is, `SubNetIO` is an extension that relies upon the functionality provided by `CampusIO`. As such, it is appropriate to run `CampusIO` or `SubNetIO`, but not both, since `SubNetIO`’s functionality is a superset of the `CampusIO`’s functionality.

Figure 6 is a simplified representation of the logic of `CampusIO` in perlish pseudo-code.

One of the features unique to the FlowScan system is its modular reporting structure. Because most of what the system does is the responsibility of peripheral modules, the `flowscan` script essentially just

provides a framework for arbitrary testing and periodic reporting on flow content.

Stateful Inspection of Flows

The `CampusIO` report uses a number of heuristics that help it to identify elusive traffic, such as that of the Napster application [Plonka] or of PASV mode ftp file transfers. These heuristics employ a method of *stateful inspection*, which is essentially a variation of the stateful inspection of packets employed by many modern firewalls. Such firewalls track the state of an application session by observing information within a *packet* or series of *packets*. This technique enables the firewall to filter packets according to whether or not a session has been established and is still active. This state information is gleaned via passive inspection of either the packet header or application payload.

Similarly, FlowScan attempts to track the state of an application session, or series of sessions, by observing the information within *flows*. This flow-based stateful inspection enables counters to be maintained and reported upon for flows which would otherwise be left unidentified or be misidentified. For example, the identifying of traffic using a simple test of protocol and port number (i.e., tcp and port 6699) to identify Napster data flows is likely to sometimes erroneously match flows from applications such as PASV mode ftp which negotiate dynamic, unprivileged

```
package CampusIO;

sub perfile {
    # Remember the time-stamp from the filename.
    whence = filename2time_t(filename)
}

sub wanted {
    if (exporter_hop(flow::next_hop)) {
        # This flow is destined for another "local" flow-exporting router.
        # We'll catch it later, if and when it's exported by that router.
        return 0
    }

    if (outbound_hop(flow::next_hop) or outbound_interface(flow::output_if)) {
        # This flow is outgoing.
        outbound_total++
    } elsif (inbound_address(flow::destination_address)) {
        # This flow is incoming.
        inbound_total++
    } else {
        # This flow (an "intranet" flow) is unwanted.
        return 0
    }

    return 1
}

sub report {
    update_RRD_files(whence, inbound_total, outbound_total)
}
```

Figure 6: The logic of `CampusIO`.

port numbers. We minimize or eliminate this error by employing a more complicated test based on stateful inspection.

Figure 7 is a pseudo-code example of the logic used to detect Napster flows and PASV mode ftp data flows, which demonstrates the technique.

Also, without this sort of analysis, a large percentage of our campus traffic would be simply labeled as “unknown”. As it is now, still more than 30% of our campus traffic remains unclassified.

FlowScan is not the only package that performs such stateful inspection of flows. Although FlowScan has for some time performed such inspection to identify Real Media flows and Napster flows, such a technique was developed independently by Simon Leinen for Fluxoscope [Fluxoscope] to identify PASV mode

ftp data flows, a feature only more recently added to FlowScan.

Life with FlowScan

FlowScan produces a variety of graphs that provide a different view of network traffic than that provided by most other tools. For example, flow-per-second graphs are often as useful in network management as are packet-per-second and byte-per-second (bandwidth) graphs. FlowScan’s RRD files are configured in such a way that they aggregate counters into less granular totals over time. For instance, five minute samples are combined into 30 minute samples, then into two hour samples, then into 24 hours samples. As a result, the sort of anomalies which we are able to discover is determined by the time length of the graph. In general we have found that by graphing over both

```
sub Napster_wanted {
  if (ICMP != flow::protocol || not (TCP == flow::protocol and
    (1024 < flow::srcport and 1024 < flow::dstport)) {
    return 0
  }
  # flow is either ICMP or TCP on unprivileged ports
  if (inbound(flow)) {
    direction = 'in';
    outside_addr = flow::srcaddr;
    inside_addr = flow::dstaddr;
  } elsif (outbound(flow)) {
    direction = 'out';
    outside_addr = flow::dstaddr;
    inside_addr = flow::srcaddr;
  }
  if (TCP == flow::protocol and ACK & flow::tcp_flags and
    is_napserver(outside_addr)) {
    # flow involves an outside host that is a "publicly advertised" napserver
    remember_napster_server(outside_addr, flow::endtime);
    remember_napster_user(inside_addr, flow::endtime)
  } elsif (is_napuser(inside_addr)) {
    # flow involves an inside host that has talked to a napserver recently
    if ((TCP == flow::protocol and
      napster_ports(flow::srcport, flow::dstport)) or
      (ICMP == flow::protocol and 28 == flow::bytes/flow::pkts)) {
      # Confidence is high that this is a Napster application flow because
      # flow is either TCP on Napster "default" ports or
      # flow is ICMP using the "known" Napster ICMP packet size.
      napster_total++;
      return 1
    } else {
      # Confidence is lower that this is really a Napster application flow
      # because the port numbers are not Napster defaults or the ICMP
      # packet size isn't right. We'll keep a count of these anyway.
      maybe_napster_total++
    }
  }
  return 0
}
```

Figure 7a: Pseudo-code logic to detect Napster flows and PASV mode ftp data flows, part 1.

short and *long* terms to be useful. However, each serves its own purposes.

Short-Term Analysis

By default, FlowScan's window for short-term analysis is 48 hours, that allows easy comparison of those two consecutive days. The standard graphs

supplied with FlowScan provide views of traffic by network or subnet, by application or service, and by Autonomous System over the past 2 days. In each of these three graph categories, graphs are available by bits-per-second, packets-per-second, and flows-per-second.

```

sub ftp_PASV_wanted {
    if (TCP != flow::protocol) {
        return 0
    }
    # flow is TCP
    if (not (21 == flow::srcport or 21 == flow::dstport or
        (1024 <= flow::srcport and 1024 <= flow::dstport))) {
        return 0
    }
    # flow is either ftp (control port 21) or
    # TCP on unprivileged ports (>= 1024)
    if (1024 <= flow::srcport and 1024 <= flow::dstport) {
        # could be ftp PASV data flow
        if (ftp_session_exists(flow::srcaddr,
                               flow::dstaddr,
                               flow::endtime)) {
            # flow is probably ftp PASV data
            ftp_PASV_total++
        }
        return 1
    }
    # flow is ftp (control port 21)
    if (ACK & flow::tcp_flags) {
        # ftp (control port 21) stream is active
        remember_ftp_session(flow::srcaddr, flow::dstaddr, flow::endtime)
    }
    if (FIN & flow::tcp_flags) {
        # ftp (control port 21) stream is closed
        remember_ftp_session_closed(flow::srcaddr, flow::dstaddr, flow::endtime)
    }
    # this flow wasn't ftp PASV data
    return 0
}

sub perfile {
    # ...
    # Forget Napster server hosts (in the outside world) and user hosts
    # (on the inside) that haven't talked Napster recently.
    forget_quiet_napservers(30*60);
    forget_quiet_napusers(30*60);
    # Forget ftp session if we have seen FIN and 15 mins has passed
    # since we don't expect DATA flows so long after control port 21
    # has closed.
    # Also, forget ftp session if we haven't seen ACK on its control
    # port 21 in the past 30 mins since it's likely to have timed
    # out by now. Perhaps we missed the flow which indicated that
    # the control port 21 stream has closed.
    forget_closed_or_quiet_ftp_sessions(15*60, 30*60);
    # ...
}

```

Figure 7b: Pseudo-code logic to detect Napster flows and PASV mode ftp data flows, part 2.

The graphs over a short, recent time frame are based upon the data that FlowScan keeps at five-minute intervals. The coarser-grained averaging in the longer term graphs will often hide anomalies seen in these finer-grained graphs. Network abuse, such as flood-based Denial of Service attacks, are easily visible.

Specifically, in our experience with FlowScan, we have learned that a discrepancy between the number of inbound and outbound flows or packets is an indication of abusive traffic, such as a DoS flood. Sudden changes in packet counts, especially when constrained to one protocol, are usually indications of a flood as well. Figure 8 is an example graph based on flow counts representing a flood of traffic which was unnoticeable on bandwidth usage graphs and nearly so on packet count graphs. The traffic responsible for the spikes in TCP flows was a flood of incoming 40-byte TCP ACK packets to which the campus host to which they were directed responded with 40-byte TCP RST packets. Even though a flood of small packets with dynamic source addresses produces spikes of equal magnitude in both packet and flow graphs, it is more readily noticed in the flow graph because the spike is a much larger proportion of the total flows than it is of the total packets.

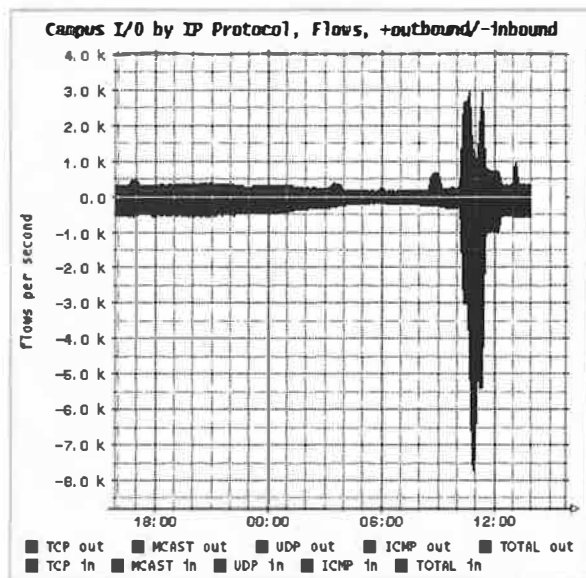


Figure 8: Graph produced 2000/09/24 showing a 40-byte TCP DoS flood.

When visualizing over the short term, it is important to remember that FlowScan increments traffic counters at the time *when the flows are exported*. More precisely, it records the values of these counters with the time-stamp corresponding to the five-minute interval in which cflowd wrote the flow to the raw flow file. As such, the time reported in the graphs is some function of the flow end time, but may involve timeouts defined in the NetFlow implementation. This

time-stamp is thus not necessarily that when the represented traffic was actually observed by the router.

It is possible for a FlowScan graph to report a quantity of traffic in a given five-minute period that exceeds that actually forwarded by the router. On occasion, the quantity reported could even exceed the physical capacity of the link that carried it, which can be misleading. The reason that FlowScan operates in this way is twofold. First, for the sake of simplicity, FlowScan assumes that the traffic represented there-in must have occurred within the given five minute period. Without this assumption, we couldn't simply accumulate totals, and plot those values vs. time. Second, while NetFlow flows contain start and end time information for the packets in a given flow, they do not contain any hint as to how the delivery of those packets was distributed between the flow start and end time. So, even if FlowScan attempted to record the real time at which traffic was observed, it would not necessarily be any more accurate. We will see that the effect of this "time kludge" is negligible as data is coalesced into less granular time samples and visualized in longer-term analyses.

Long-Term Analysis

When producing FlowScan graphs, one may simply specify the number of *hours* over which to plot, such as 24 (hours) times 365 (days). Increasing the hours significantly beyond the default becomes an exercise in customization, since these graphs often need to be annotated with dates and other descriptions so that they have meaning to their intended audience.

Daily averages, which are used when graphing anything but fairly recent time ranges, hide spurious abuse activity such as flood-based Denial of Service attacks because such attacks are usually short-lived and will be severely minimized by averaging. As such the primary value of graphing over extended periods with daily averages is to aid in capacity planning and perhaps to help target traffic shaping efforts.

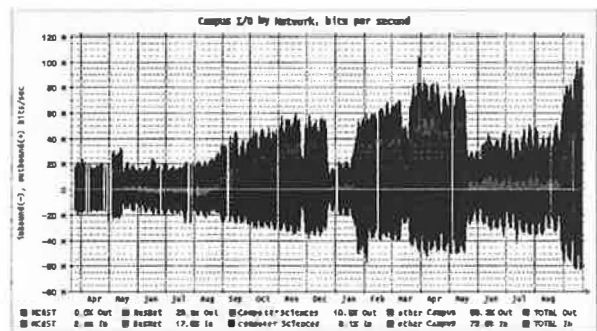


Figure 9: Graph produced 2000/09/21 showing campus traffic by network over the past 550 days.

Figure 9 shows a sample FlowScan long-term graph using daily averages over 550 days.

Information gleaned from this graph includes:

- The academic calendar dramatically influences the traffic levels, but only with respect to traffic

to and from ResNet, the residence halls network.

- There has been an increase in outbound ftp traffic from the Computer Sciences department within the past year.
- While our outbound traffic level has consistently exceeded our inbound traffic level, the discrepancy between the two appears to be increasing.

Custom Graphs

In addition to using the “canned” graphs supplied with the FlowScan distribution, FlowScan users can produce custom graphs with the companion tool named RRGraher [RRGraher]. RRGraher is the “Round Robin Graph Construction Set” and is

implemented as a single perl CGI script, which runs under a web server that has access to RRD files. RRGraher is also general front-end for RRDtool that allows users to interactively build graphs of their own design from any RRD files. Since it is based on RRDtool, not FlowScan, it allows one to generate graphs containing data from RRD files containing data from any RRDtool-based systems such as MRTG, Cricket, and FlowScan.

Figure 11 is a sample graph produced using RRGraher. This graph shows the bandwidth used by ftp data transfers from ftp servers to ftp clients both in and out of the campus as determined by FlowScan. Figure 10 is the resulting RRDtool command that RRGraher wrote and executed internally to produce

```
rrdtool graph /your/file/name/here.gif \
--start 'Aug 15, 2000' \
--end 'Sep 15, 2000' \
--vertical-label 'bits/sec' \
--title 'ftp PASV and ftp-data, +outbound/-inbound' \
'DEF:A=ftp-data_src.rrd:in_bytes:AVERAGE' \
'DEF:B=ftp-data_src.rrd:out_bytes:AVERAGE' \
'DEF:C=ftpPASV_src.rrd:in_bytes:AVERAGE' \
'DEF:D=ftpPASV_src.rrd:out_bytes:AVERAGE' \
'CDEF:E=A,8,*,-1,*' \
'CDEF:F=C,8,*,-1,*' \
'CDEF:G=B,8,*' \
'CDEF:H=D,8,*' \
'COMMENT:A) ftp-data_src AVERAGE in_bytes' \
'COMMENT:\n' \
'COMMENT:B) ftp-data_src AVERAGE out_bytes' \
'COMMENT:\n' \
'COMMENT:C) ftpPASV_src AVERAGE in_bytes' \
'COMMENT:\n' \
'COMMENT:D) ftpPASV_src AVERAGE out_bytes' \
'COMMENT:\n' \
'AREA:E#0000FF:ftp-data src in (A,8,*,-1,*)' \
'GPRINT:E:MIN:(min=%.11f%S' \
'GPRINT:E:AVERAGE:ave=%.11f%S' \
'GPRINT:E:MAX:max=%.11f%S)' \
'COMMENT:\n' \
'STACK:F#00FFFF:ftp PASV src in (C,8,*,-1,*)' \
'GPRINT:F:MIN:(min=%.11f%S' \
'GPRINT:F:AVERAGE:ave=%.11f%S' \
'GPRINT:F:MAX:max=%.11f%S)' \
'COMMENT:\n' \
'AREA:G#00FF00:ftp-data src out (B,8,*)' \
'GPRINT:G:MIN:(min=%.11f%S' \
'GPRINT:G:AVERAGE:ave=%.11f%S' \
'GPRINT:G:MAX:max=%.11f%S)' \
'COMMENT:\n' \
'STACK:H#A0522D:ftp PASV src out (D,8,*)' \
'GPRINT:H:MIN:(min=%.11f%S' \
'GPRINT:H:AVERAGE:ave=%.11f%S' \
'GPRINT:H:MAX:max=%.11f%S)' \
'COMMENT:\n'
```

Figure 10: RRDtool command created by RRGraher.

the graph. At the user's option, RRGrapher displays this command when it produces the graph, allowing the user to cut and paste it elsewhere, perhaps to schedule it as a batch or cron job.

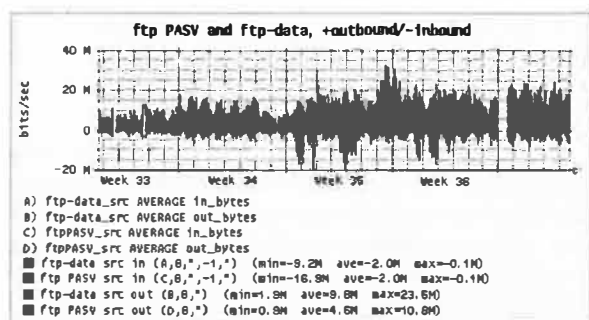


Figure 11: Graph showing campus ftp server traffic Aug 15, 2000 through Sep 15, 2000.

FlowScan Problems

FlowScan, while still extremely useful in most production networks, has exhibited a number of limitations and illuminated a number of problems. These may be instructive in the building of next generation flow analysis tools.

FlowScan's near real-time processing lags behind when processing flow files containing mostly "pathological" flows, such as those flows which represent only one packet per flow. These degenerate flows are produced to represent the flood of traffic produced during most Denial of Service attacks because the source or destination information is forged so that each packet contains a different IP address or port number than that which preceded it. In these situations the flow export rate approaches the router's packet forwarding rate, resulting in an avalanche of flows directed to the collection host. Figure 8 is an example graph of such traffic. Unless such traffic eventually ceases within hours or is blackholed by the network administrators, FlowScan would fall hopelessly behind, since it cannot process flows at rates approaching the packet forwarding rate of high performance routers with high capacity links.

Furthermore, under "normal" circumstances on some networks, FlowScan can still become buried in data. Even with the orders-of-magnitude reduction in the number of flows vs. the numbers of packets that those flows represent, flow processing tools may be overrun with too many flows to process in close to real-time on fast links. Currently, with Cisco's version 5 flow-export, FlowScan might not be able to scale beyond monitoring a couple fully-utilized OC3 (155 Mb/s) links. As technologies such as Gb ethernet and optical switching are deployed in border routers, the packet forwarding rate at these aggregation routers could outpace the performance gains in commodity personal computer hardware that are important for flow post-processing. Specifically, the performance

adequacy of the processor and bus to persistent storage is in question. So, as packet forwarding performance improves, FlowScan-like processing may need to move from the network's DMZ to a location much closer to the end-user. This migration will require the deployment of arrays of measurement equipment. Managing an array of such devices will present new challenges for the network administrator.

In part, for tools other than FlowScan, these backlog issues have been avoided by aggregating totals on the router itself and then exporting those totals less frequently as summary PDUs. Another alternative is to adopt SNMP polling-based gathering of flow statistics rather than collecting unsolicited, exported flow data. However, both of these methods prevent nearly all the interesting post-processing that FlowScan currently performs, and also eliminate the archivable record of network traffic that detailed flow PDUs provide, which has proven invaluable during investigations of security compromises and network abuse.

Other problems occur when either NetFlow export or FlowScan is misconfigured. In complicated environments, configuring Cisco routers for cflowd has been easy to do incorrectly, especially since ip route cache flow should be enabled only on the appropriate interfaces. Misconfiguration can result in missing, skipped, or duplicated flows. Obviously, such misconfiguration may result in inaccurate flow statistics which, unfortunately, are dutifully stored into RRD files by FlowScan. These files are FlowScan's sole means of keeping archive data to answer historical questions or examine long-term trends. To combat this problem, in our use of FlowScan, we maintain a diary of interesting events for correlation with spurious graph anomalies.

Finally, identifying flows as representing traffic for specific applications is becoming increasingly difficult. Traditionally, the packets and flows of "well-known" applications were easily identified by interrogating values in the IP header such as port number. However, this identification is complicated by the increasingly popular technique of dynamically negotiating ports for connections, as well as the use of unreserved port numbers by many modern applications. Since flows do not contain the packet payload, we are not always able to label a given flow by application name with a high level of confidence, but instead must resort to compromises and heuristics. Furthermore, as VPNs, tunneling, and encryption become more common, network applications will intentionally hide the payload and even the IP header. Quality-of-Service initiatives may address some of these passive measurement problems since the same criteria by which packets will be marked for QoS purposes may be sufficient for flow identification as well. The identification of a flow's user and service class may have to suffice.

Future Directions and Possibilities

Alerts

Automatic event notification or *alert* capability would be a useful addition to FlowScan. A report module named “CampusIOAlert” is currently under development which dispatches alerts to email addresses or alphanumeric pagers. We have focused on alerts based on tests that would be unlikely to generate false positives and would not need to be executed more than once per flow file. For instance, we have implemented an alert to report flood-based Denial of Service attacks based simply on the detection of an imbalance of inbound versus outbound flows.

The opportunity for alert modules to perform tests on individual flows may also be useful, although more processing intensive since such tests must be executed once per flow rather than once per flow file. For example, we have had good luck in identifying hosts that were infected with a specific email-attachment worm because this worm attempted to negotiate an HTTP connection from the infected host to a well-known destination host.

Ultimately, with a report such as CampusIOAlert, FlowScan users will be able to write their own alert tests using the expressive syntax of the perl language itself. These alerts will also be able to test values from RRDtool’s `rrdtool graph` command, which has expressive RPN expression evaluation and test features of its own.

Performance Enhancements

One of the largest challenges has been dealing with flow file backlog created whenever the quantity of flows exported by the router, and subsequently written to disk, exceeds the quantity that FlowScan is able to process in near real-time.

Converting FlowScan into a multi-threaded, or simply multi-process, application was thought to be a viable solution to the backlog issue, and was recently implemented by Alexander Kunz, an ambitious FlowScan user. This multi-process flowscan enlists the help of another flowscan process whenever FlowScan had more than one flow file yet to be processed. A mutex technique in which each process takes a numbered “ticket” was employed to ensure that subsequent updates of the RRD files occurred in the proper time-series order.

At the time of this writing, September 2000, the challenge of integrating this multi-process patch into the mainline FlowScan distribution has been delayed in favor of simply improving the performance of the single-threaded model. A faster single-threaded solution avoids those complications inherent in multi-threading which we have yet to address, namely that the FlowScan code would need to enforce serialized access to its internal data structures used for stateful inspection. Also, light-weight thread support in perl, while available, is still considered experimental.

Instead, to vastly improve FlowScan’s single-process performance, IP address prefix matching was reimplemented with Patricia Trie lookups in a C language extension to perl – i.e., a perl module. The term “Trie” is derived from the word “retrieval” but is pronounced like “try”. Patricia stands for “Practical Algorithm to Retrieve Information Coded as Alphanumeric” and is thoroughly described in [WrightS]. The Patricia Trie performance characteristics are well-known as it has been employed for routing table lookups within the BSD kernel since the 4.3 Reno release [Sklower]. We chose the Patricia data structure and search algorithm upon realizing that FlowScan’s decisions based upon IP addresses are nearly identical to those decisions that an IP router must make when delivering a packet based upon destination IP address.

Multi-vendor Compatibility

While nearly all the focus so far has been on Cisco’s NetFlow feature, other network hardware vendors have developed similar flow-based accounting technology. Juniper, with its announcement of its JUNOS software release 4.1R1 in August 2000, now supports a sub-set of the accounting functionality using the Cisco NetFlow-defined PDU formats. Riverstone Networks, formally a portion of Cabletron, has a protocol called LFAP: Lightweight Flow Accounting Protocol. Related software, such as slate, is available at [NMops].

We are looking at the possibility of supporting flow processing based on the implementations of these or other vendors. Potentially one could build this support for FlowScan by enhancing `cflowd` to support other flow export implementations, by modifying other collectors to produce the `cflowd`-defined raw flow file format, or by modifying FlowScan or the underlying Cflow perl module to be able to process a stream of flows of a different format.

Availability

FlowScan is freely available under the terms of the GNU General Public License [FlowScan, GPL]. FlowScan-1.002, that was released March 21, 2000, was used at approximately 100 sites. FlowScan-1.003 was released September 15, 2000. More information on FlowScan is available at <http://net.doit.wisc.edu/~plonka/FlowScan/>. A mailing list for flowscan users is archived at <http://net.doit.wisc.edu/~plonka/list/flowscan/>.

Likewise, RRGrapher is freely available under the same terms at <http://net.doit.wisc.edu/~plonka/RRGrapher/>.

FlowScan’s primary building blocks, `cflowd` and RRDtool were both developed with the support of CAIDA. They are freely available at <http://www.caida.org/tools/measurement/cflowd/> and <http://ee-staff.ethz.ch/~oetiker/webtools/rrdtool/>, respectively.

Summary

FlowScan has become a useful tool for our network engineering team. We have run it nearly

continuously to measure and analyze all traffic to and from our campus since late 1998. The resulting reports and graphs have been of use for the detection of abuse and other anomalies that are detrimental to the network backbone's performance as a whole. Without such a tool, these incidents would have gone unexplained. Also, FlowScan's long-term graphs have been used to convey the impact of Napster to our management, and to justify progressive upgrade and bandwidth acquisition plans for the campus. With its public release, FlowScan has proven useful for these and other purposes by a user base of educational institutions, government institutions, corporations, and Internet service providers.

When attempting to meet the challenges of managing a heavily utilized IP network, near real-time traffic analysis and visualization quickly becomes an essential technology. One way to provide these capabilities is by utilizing Internet traffic flow profiling based on technology available in most networking equipment. FlowScan is a system designed to provide this analysis continuously in near-real time and can be an effective tool to better understand Internet traffic.

Acknowledgements

The initial public FlowScan, released in September 22, 1999, was patiently tested by a number of volunteers. The effort, feedback, and encouragement of those users and of members of the flowscan mailing list, has been much appreciated and has made FlowScan a better tool. James Deaton of OneNet, Ted Frohling of the University of Arizona, John Kristoff of DePaul University, Gregory Goddard of the University of Florida, and others have made either a private or public FlowScan-based web site accessible to myself and other FlowScan users. This information has been invaluable for research and development. Alexander Kunz of Nextra in Germany and Michael Hare of the University of Wisconsin-Madison have made tangible contributions to FlowScan. Alexander hacked out the first multi-threaded FlowScan, prototyped a visualization enhancement for the graphs, and made the changes available to the user community. Michael enthusiastically began development of the aforementioned CampusIOAlert report module. Daniel McRobb, Tobi Oetiker, and CAIDA have provided the main tools upon which FlowScan is built, namely cflowd and RRDtool. Thanks to K. Claffy, of the Cooperative Association for Internet Data Analysis (CAIDA), and Denis DeLaRoca, of the University of California, Los Angeles who have helped by providing their thoughts and encouragement on this and related work.

Author Information

Dave Plonka has developed a number of free software packages, many of which are network management tools. He works as a systems programmer doing network engineering within the Division of

Information Technology (DoIT) at the University of Wisconsin – Madison. In the formative years of his working life he was a programmer in the commercial software industry focusing on the development of portable libraries and relational database applications under VMS and Unix. In 1991, he received a B.S. in Computer Science from Carroll College in Waukesha, Wisconsin. Dave can be reached at plonka@doit.wisc.edu or via <http://net.doit.wisc.edu/~plonka/>.

References

- [ClaffyPB] K. Claffy, G. C. Polyzos, and H.-W. Braun, "Internet traffic flow profiling", UCSD TR-CS93-328, SDSC GA-A21526, <http://www.caida.org/outreach/papers/itf.html>, November 1993.
- [NetFlow] Simon Leinen, "FloMA: Pointers and Software, NetFlow", <http://www.switch.ch/tf-tant/floma/software.html#netflow>.
- [MRTG] "MRTG", <http://ee-staff.ethz.ch/~oetiker/webtools/mrtg/>.
- [Oetiker] Tobias Oetiker, "MRTG – The Multi Router Traffic Grapher", USENIX LISA '98 Conference Proceedings, 1998.
- [Cricket] "Cricket Home", <http://cricket.sourceforge.net/>.
- [Allen] Jeff R. Allen, "Driving by the Rear-View Mirror: Managing a Network with Cricket", USENIX NETA '99 Conferences Proceedings, 1999.
- [RRDtool] "RRDtool – Round Robin Database Tool", <http://ee-staff.ethz.ch/~oetiker/webtools/rrdtool/>.
- [Plonka] Dave Plonka, "UW-Madison Napster Traffic Measurement", 2000., <http://net.doit.wisc.edu/data/Napster/>.
- [Cisco] "Cisco's IOS NetFlow feature", <http://www.cisco.com/warp/public/732/netflow/>, http://www.cisco.com/warp/public/cc/cisco/mkt/ios/netflow/tech/napps_wp.htm.
- [McRobb] Daniel W. McRobb, "cflowd configuration", <http://www.caida.org/tools/measurement/cflowd/configuration/configuration.html>, 1998-1999.
- [cflowd] "cflowd – CAIDA's flow analysis tool", <http://www.caida.org/tools/measurement/cflowd/>.
- [McRobb2] Daniel W. McRobb, "cflowd design", <http://www.caida.org/tools/measurement/cflowd/configuration/design/design.html>, 1998.
- [patch] "patches to cflowd", <http://net.doit.wisc.edu/~plonka/cflowd/>.
- [Fluxoscope] Simon Leinen's "Fluxoscope", <http://www.switch.ch/lan/stat/fluxoscope/>.
- [Leinen] Simon Leinen, "Fluxoscope: a System for Flow-based Accounting", 2000., <http://www.tik.ee.ethz.ch/~cati/deliv/CATI-SWI-IM-P-000-0-4.pdf>.
- [RRGrapher] Dave Plonka, "RRGrapher – the Round Rober Grapher, a Graph Construction Set for

- RRDtool”, <http://net.doit.wisc.edu/~plonka/RRGrapher/>.
- [WrightS] Gary R. Wright, W. Richard Stevens, *TCP/IP Illustrated, Volume 2: The Implementation*, Addison-Wesley Publishing Company, Inc., 1995.
- [Sklower] Keith Sklower, “A Tree-Based Packet Routing Table for Berkeley UNIX”, *Proc. USENIX Conference Proceedings*, <http://www.cs.berkeley.edu/~sklower/routing.ps>, 1991.
- [NMOps] “Network Management Operations”, <http://www.nmops.org/>.
- [FlowScan] “FlowScan”, <http://net.doit.wisc.edu/~plonka/FlowScan/>.
- [GPL] “GNU General Public License” Version 2, Free Software Foundation, Inc., <http://www.gnu.org/copyleft/gpl.html>, 1991.

Tracing Anonymous Packets to Their Approximate Source

Hal Burch – Carnegie Mellon University
Bill Cheswick – Lumeta Corp.

ABSTRACT

Most denial-of-service attacks are characterized by a flood of packets with random, apparently valid source addresses. These addresses are spoofed, created by a malicious program running on an unknown host, and carried by packets that bear no clues that could be used to determine their originating host. Identifying the source of such an attack requires tracing the packets back to the source hop by hop. Current approaches for tracing these attacks require the tedious continued attention and cooperation of each intermediate Internet Service Provider (ISP). This is not always easy given the world-wide scope of the Internet.

We outline a technique for tracing spoofed packets back to their actual source host without relying on the cooperation of intervening ISPs. First, we map the paths from the victim to all possible networks. Next, we locate sources of network load, usually hosts or networks offering the UDP chargen service [5]. Finally, we work back through the tree, loading lines or router, observing changes in the rate of invading packets. These observations often allow us to eliminate all but a handful of networks that could be the source of the attacking packet stream. Our technique assumes that routes are largely symmetric, can be discovered, are fairly consistent, and the attacking packet stream arrives from a single source network.

We have run some simple and single-blind tests on Lucent's intranet, where our technique usually works, with better chances during busier network time periods; in several tests, we were able to determine the specific network containing the attacker.

An attacker who is aware of our technique can easily thwart it, either by covering his traces on the attacking host, initiating a "whack-a-mole" attack from several sources, or using many sources.

Introduction

One of the major problems on the Internet today is denial of service (DoS) attacks against machines and networks. As opposed to other types of attacks, DoS attacks attempt to limit access to a machine or service instead of subverting the service itself. DoS attacks are simple to design and implement, and there is a plethora of readily available source code which will perform the task. DoS attacks send a stream of packets at a victim that swamps his network or processing capacity, denying access to his regular clients.

There are two basic targets of DoS attacks: machines and networks. SYN attacks [11] are an example of an attack against a machine. In these attacks, a series of TCP SYN packets are sent to a host, filling its table of "half-open" TCP connections. Normal connection attempts are dropped. The basic problem with a skillfully run SYN attack is that the clients and the attackers are indistinguishable without further processing. The server must issue SYN/ACK packets and wait for the client to respond. This particular attack can be mitigated with appropriate algorithms in the server [11]. Other machine attacks may be more difficult to defend against.

The second target type, networks, are much more difficult to defend. Here, the goal is to overload a

company's connection to its ISP. The attacker focuses a large stream of data towards the company's network, often from a number of sites. The company's connection becomes congested, resulting in packet loss. Since routers cannot distinguish between attacking packets and valid client packets, they drop them with equal probability. If the attacker can send packets fast enough, the drop rate can become so high that an insufficient number of a client's packets get through. Thus, clients cannot get reasonable service from any machine beyond the loaded link. The most common of this type of attack is the Smurf attack [8], although recent distributed denial of service attacks (DDoS) [9] have been of this flavor.

The major advantage of DoS attacks is that it is quite difficult to determine the actual source of the attack. Since the attacker can basically put any packet on the local wire, the attacker creates packets whose source IP address is invalid and completely random. Thus, when the victim receives these packets, they are unable to determine the source. The current technique for tracing a packet stream back to the source requires cooperation of all the intervening ISPs. This is something that is difficult to obtain, since the victim is rarely a customer of all of the ISPs between it and the attacker. The standard technique will be discussed in more detail later.

We have developed a method to trace a steady stream of anonymous Internet packets back towards their source. The method does not rely on knowledge or cooperation from intervening ISPs along the path. In addition, tracing an attacking stream requires only a few minutes once the system is set up for a victim.

Basic Technique

We begin by creating a map of the routes from the victim to every network, using any known mapping technology [1, 6, 7]. Then, starting with the closest router, we apply a brief burst of load to each link attached to it, using the UDP chargen service [5]. If the loaded link is a component of the path of the attacking stream, our induced load will perturb the attacking stream. Thus, if the stream is altered when we load a link, this link is probably along the path from the source host of the attack to the victim host. If the intensity of the stream is unperturbed by the load, it is unlikely that the stream of attacking packets is utilizing that link, so we do not need to examine the networks “behind” that link.

We continue working back through the network router by router, pruning branches that do not perturb the attack, as we try to narrow the attack source to one network, at which point we can shift to more standard traceback methods by contacting the entity which controls that network.

Executing a trace effectively does require significant preparation in the way of data collection. We need to collect network data, as well as traceroutes from the victim to all possible networks. Due to asymmetric routes, naively, directional data must be collected and maintained by reverse traceroute servers or other means in order to have perfect data. We collect outbound paths and assume that the incoming paths are approximately the reverse of those paths. While this is not completely accurate, by collecting the paths to all networks, we can determine what links could be used on a path from a given network to the victim's network, so this assumption does not cause as many inaccuracies as might otherwise occur.

Because we need to induce isolated load on specific network segments that are not in our purview, we must identify sources “willing to” (read: will) perform that task. We recognize that ISPs are now quite regularly turning off the services that we exploit to induce these loads. Thus, we must identify cooperative hosts at the right places in our network map in order to do produce the required load.

This element of the technique is worrisome, since it constitutes a brief denial-of-service attack on that network link. Hackers already employ bulk versions of this approach for denial-of-service attacks. Our technique, on the other hand, carefully limits load to segments only long enough to rule them out as a possible component of the suspected path. The difference is analogous to that between a sword and a scalpel.

In any case, we recognize the antisocial aspect of this technique, and expect that the tool will be used rarely and only in appropriate situations. Possible users include law enforcement, the military, ISPs, and companies policing their own private intranets.

Before attacks or victims are even known, a trusted machine must develop and maintain a current database of networks and load generators. The current version of the tool executes the trace from the victim (targeted) network, but a sufficient complete map of the Internet might allow a neutral third party to run the detecting utility, which would allow flexibility in where to spread some of the bandwidth cost of the tool.

In either case, the tracing machine emits packets that stimulate traffic flow through a desired router or link. A visual display of various statistics of the incoming packets on the victim's network helps determine if that link is used by the packets.

An operator using a tool to probe links on the path back to the attacker. The application of load is done manually (see Figure 1). Though there are algorithms that might automate this process, we require human intervention to reduce the cost of programming errors. We try to supply the operator with information about the amount of load she is inflicting on networks, and she can choose to stop using packet-source networks that have already generated a lot of load.

If the induced load is sufficient to induce drops of incoming packets, it quickly and dramatically affects the attacking flow. The discomfort to ISPs and end users is brief enough that it is likely to escape notice. If the load does not induce loss, it may be necessary to run the load generators longer and seek more subtle effects on the workload.

Our technique appears to work better when the network is already heavily loaded, though one can imagine more subtle statistical effects that may be detectable when the Internet is relatively quiet. Our attempts to discover such effects has met with little success. We found we were interacting with cache and other optimizations in various routers. In some cases, our applied load actually increased the packet attack rate!

Assumptions

Our technique does rely on several assumptions, but our experience indicates they are often valid and the technique can work.

Assumptions About the Internet

We assume that most routes over the Internet are symmetric. Asymmetric routes confuse our mapping, traceback and loading. However, the proliferation of reverse traceroute servers, which has proven quite useful for network diagnosis and debugging, might also facilitate construction of at least a partial directional map of routes.

We also assume that we can generate enough load on a particular Internet link to affect performance, in particular loss, statistics of the stream of attacking packets. We must have access to enough packet generators beyond the tested link to load it, which can be challenging across infrastructure with fast links and slower downstream networks. The techniques for doing this will be discussed below.

Hacking Behavior

We assume that the attack is from a single host, at a fairly consistent rate, and runs for a reasonably long time. Denial-of-service attacks are more vexing if they are ongoing, and we have seen attacks that last for weeks. We have seen attack rates of 200-500 packets per second from a single host. We need time to move equipment and programs into place, map routes, and perform the actual traceback.

Bizarre behavior can occur during the traceback, so we have to examine clues carefully. For example, the operator might notice an attacking stream drops by 33% rather than dropping off entirely. Such behavior would be consistent with two or three concurrent attacks from separate hosts; it also possible that the attacking stream is being load-balanced across three

different links. Unfortunately, only one packet stream can be traced at a time, so being able to distinguish among the streams would be essential to be able to perform the trace. The operator might be able to use the arriving TTL value, assuming packets within each stream are launched with the same TTL value, and with each stream from different hop distances away. `topdump`'s filters provide the tools necessary to isolate such parameters, so that feature of the tool can be used if one of these parameters are sufficient to distinguish between streams.

We assume the attacker does not know that her packets may be traced. An effective hacker attacks from co-opted hosts and never returns to the attacking machine. She hides her trail through a thread of login sessions across many hosts and networks before attacking the target. The denial-of-service attack we target with this tool is a one-way packet flow, which does not rely on interactive login sessions.

We assume that there is something forensically interesting at the source of the attack. The effort of running our tool may not be justified if the result is just disabling one attacking host or convincing one community of computers to enforce ingress filtering

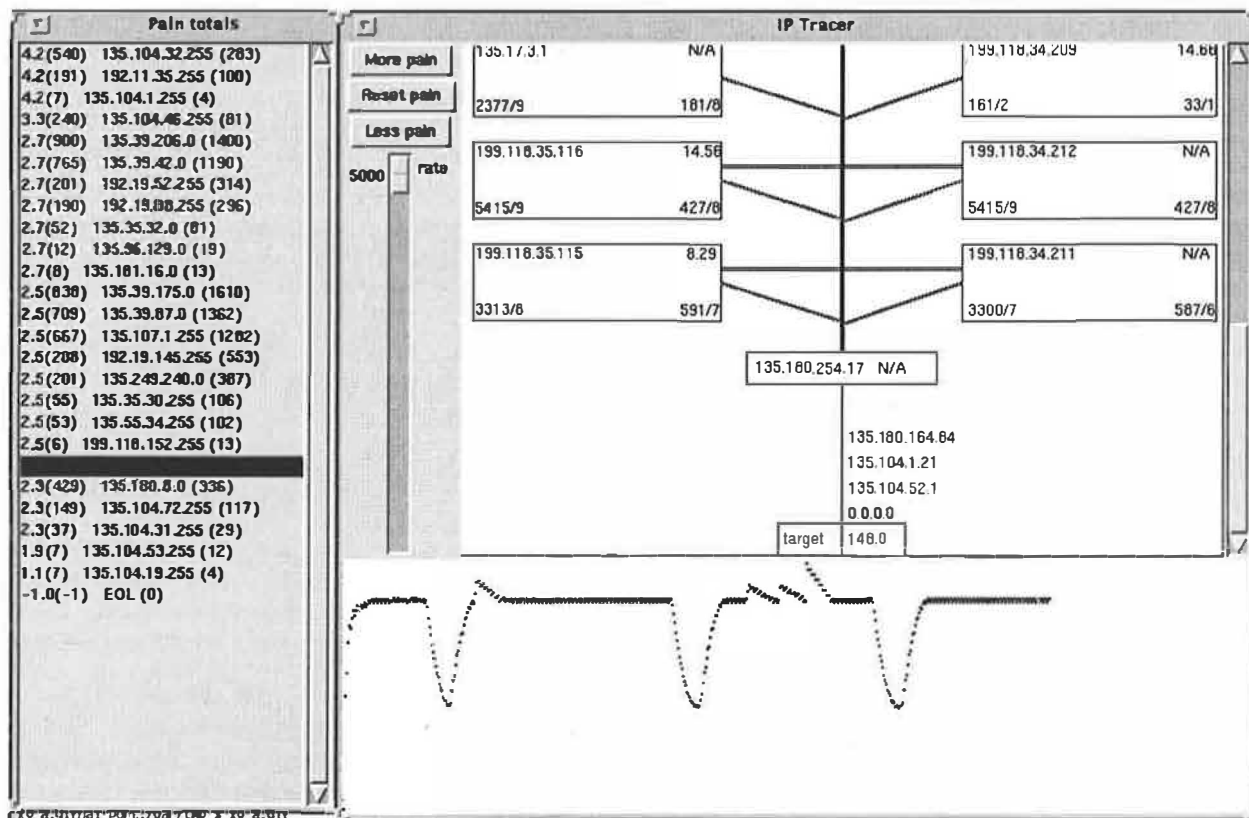


Figure 1: Screen shot of trace-back program. The left-hand screen gives information about the amount the usage of different hosts to generate pain. The bottom is a graph showing the number of packets received per second. The right-top shows the traceback step. The bottom of the traceback shows the path so far, and the top shows the possible next hops. The horizontal double lines are load-balanced lines, so these two IP addresses are really equivalent, for traceback purposes.

[10]. We may be able to catch someone who was not very cautious because he did not expect his packets to be traced. The difficulty of the tracing task renders this a common assumption of hackers.

We also assume that the attacker is unfamiliar with the techniques we provide here. These techniques are easily thwarted in several ways, including modifying the attacking program to vary the source of the attack, altering the frequency of the packets randomly, and attacking from many different sources (the “whack-a-mole” attack).

Network Load: No Gain, No Pain

Once we have determined the path to each network on the Internet, the traceback is done by walking backwards through the resulting directed graph. We load a link and hopefully cause enough packet-loss to see a noticeable drop in the rate of attacking packets. If a significant drop occurs, we can be fairly certain that the tested link is on the path from the attacker to the victim. Otherwise, either the link is not on the path or we did not provide enough load, or ‘pain,’ to that link to incur packet loss. Note that since most links are full duplex, we need to load the link in the direction towards the victim.

This traceback requires making a high capacity link very busy for a short period of time, on the order of a second. It is difficult to generate a flow of packets from a single host that will do this: it would have to come from a fast host on a fast, unloaded link. We would prefer some leverage, some “gain,” on packets we emit. If we send out a flow of x bits per second (bps), we want the resulting flow across the link to be of kx bps, where k is large enough.

To produce the load, we could send a series of messages, such as ICMP echo request (ping) packets [4], from the *victim*’s network out to distant networks whose return path we expect to include the link we wish to load. However, using ICMP echo request packets gets us only one byte in return for every byte we send out, which is a gain of only 1. In addition, the return packets traverse the entire network back to the victim, which loads the entire set of links from the assistant network to the victim, which obscures the data when trying to determine the third link out. Sending ICMP echo requests from a separate network dedicated to this service is also problematic, since the nature of Internet routing means that it is hard to assure that their return path traverses the link we are testing.

Instead of sending packets from the victim’s network, we send spoofed packets from a test host located elsewhere on the network. When testing a particular link, we send probe packets to the router on the far end of the link, using as a return address the router on the near end of the link. The near router indignantly discards the unsolicited replies (if using TCP, it actually may reset; for UDP, it may reply with a ICMP Port Unreachable).

More Gain

Many routers make special efforts to put rate limits on handling of ICMP echo requests, since they are used so often. More importantly, the gain of 1 does not help us much anyway. Thus, we need to use a different service in order to supply the load.

The most obvious choice of service to employ is the forgotten tiny service TCP character generator (chargen) [5]. This service generates continuous data to anyone who connects to it, exactly what we want. The rate of data flow is limited in general by the rate that the data is acknowledged by the client machine. At the cost of a few TCP ACKs from our side, we can coax a steady stream of data out of a site supporting this service. Several of these routed over the target link will generate substantial load. We could even use the TCP ACKs to pulse all the transmitters to provide a fine burst of load by ACK-ing several open chargen sockets simultaneously. TCP chargen is turned off on many of the Internet’s hosts and routers, but there are many that run the service, and they are easy to find.

We recognized two major problems: the TCP processing on our local host slows this chargen stream down more than we would like, and, more importantly, the chargen stream still must traverse the path all the way back to the sender, unless we try TCP sequence guessing and IP spoofing, which gets very difficult very quickly. We can circumvent this second problem by using UDP chargen instead of TCP, and spoofing the packets, but this method provides little gain, as we usually get around 102 bytes back for our 40 bytes, a gain of only 2.55. (We include 12 bytes of data in our packets that give information about the actual source of them.) The chargen RFC specifies that the return packet should have between 0 and 512 bytes of data [5] (not counting the 28 bytes for the IP and UDP headers [2] [3]). We found, however, that some Windows NT 4.0 hosts violate this standard and return up to 6,000 bytes in response to a single packet, a gain of 150!

A spoofed ICMP echo request to a broadcast address can yield gain as well. By locating networks ‘beyond’ the link to send directed broadcast ICMP echo requests to, we get a gain of one for each host on that network which responds. Unfortunately, many routers process broadcast ICMP echo requests in such a way that only the router itself returns a packet. This is, of course, fortunate for the potential victims of broadcast ICMP echo request attacks, and is, in fact, recommended for that reason [8]. However, it limits broadcast ICMP echo request’s usefulness to us.

Such routers do let other broadcast traffic through, however, and we found that we could obtain gains in excess of 200 quite often using broadcast UDP chargen packets, even on networks without NT 4.0 hosts. Surprisingly, many networks within Lucent still respond to broadcast address 0 instead of 255, so we had to check both to determine the correct one for

each network. Figure 2 shows a distribution of networks and their gain for Lucent's intranet. Note that the networks with a gain of less than 1 have a gain of 0, which means that they did not respond to broadcast UDP chargen at all.

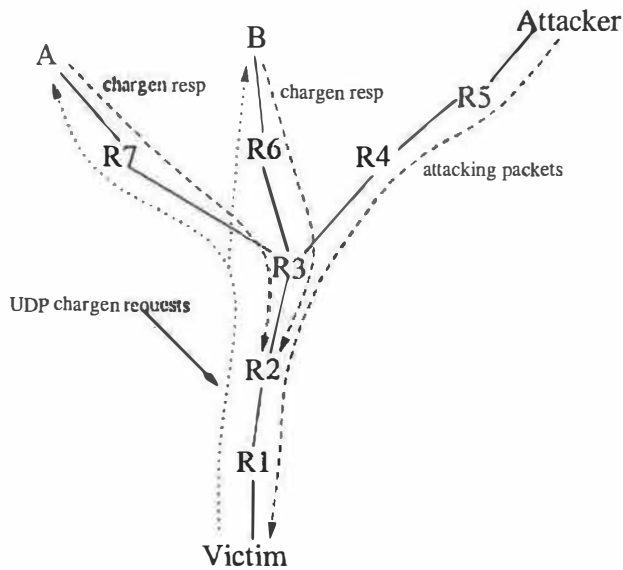


Figure 3: Example of traceback step. Packets are sent to A and B, spoofed from R2, in order to initiate packet flows towards the victim. This causes increases congestion along the R3-R2 link, which, if sufficient, will induce packet loss.

When we initiate the load, the goal is to load one line or, maybe one router. We certainly do not want to load the entire path back to the victim. We prevent this

in two different ways. First, as mentioned above, we spoof the return address of the UDP chargen packets to be the address of the router on the victim's side of the link. Second, we utilize multiple UDP chargen hosts. To test a link, we select networks that reside behind the link, as seen from the victim (see Figure 3). In particular, we select networks that have hosts that respond to UDP chargen broadcast packets. We select a network for each outbound link from the far router of the line we are testing. This strategy focuses the load on the line under examination; the packets travel to the machine over different lines, hopefully not affecting each other significantly (again, Internet routing is not inconsistent with their having traversed a common link previously in the path, though it is unusual). The load is limited by the lines the load must traverse, the speed of the networks where the load is being generated, or our ability to emit UDP chargen request packets.

The average gain seen in our experiments is around 133.8 within Lucent. One misconfigured network had a gain of several tens of thousands due to oddities in its configuration (see below). We can easily generate 2,500 40-byte packets per second, or 800 kbps. To flood a 10Mbps Ethernet only requires a gain of 12.5. Figure 4 shows the necessary gains to load a variety of line types. In order to flood a backbone link, such as an OC-48 or OC-192, one needs gains in excess of 3,000, which is larger than all but one of the gains that we have seen. However, when loading backbone links, we have help from the rest of the traffic that is traversing those links, so the actual amount of traffic required to start packet loss is much less than the number in the table. Also, we could increase the

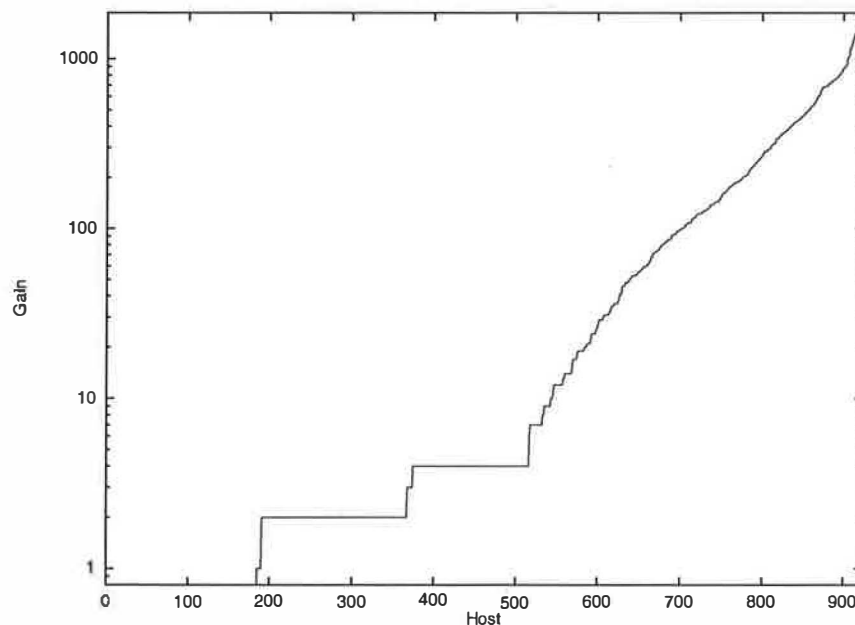


Figure 2: Distribution of gains seen using the broadcast address for Lucent's intranet. The network that generated a gain 43,509 and is excluded from this graph.

rate of outbound packets greatly by using multiple computers that connect to the Internet over different links.

Line Type	Gain Required
10Mbps Ethernet	12.5
100Mbps Ethernet	125
T1	1.9
T3	56
OC-12	777
OC-48	3,110
OC-192	12,441

Figure 4: Required gains to load a variety of line types, assuming 800 kbps of emitted packets.

Note that these numbers are a bit rough, since some of those 2,500 packets will most likely be dropped. Also, we could use 28 byte packets instead of 40 byte ones, but it is not clear that we could transmit them much more quickly.

We have discussed only one possible technique for loading the actual line; another possibility is to load the router. Diverting packet flow by sending a message directly to a router is quite difficult, as Internet backbone routers ignore various ICMP messages to redirect or stifle packet flow. Most methods to load a router have to tackle its system configuration to limit return data flow. Router designs also typically have almost all forwarding handled by a simple machine that just delegates difficult tasks to a higher layer. Less legitimate options, such as hijacking BGP sessions or breaking into the router itself are much too malicious to be seriously considered.

There are other possibilities on ways to slow routers, however. One option is to ping flood the router, i.e., send it ICMP echo requests as fast as possible. A similar alternative is to send the router a flood of packets whose Time to live (TTL) value expires at the desired hop along the path, or to transmit a stream of UDP packets to high ports to stimulate responding UDP port unreachables. Since most routers seem to rate-limit UDP port unreachable messages, we abandoned this idea before testing it extensively. The other methods do not seem to have a major effect.

Another idea is to spew packets at the router to try and upset its routing table. That is, find some sort of packet it responds regularly to (TTL exceeded, echo request) and send it a bunch of packets with random return addresses. Coping with the packets will require enough attention to unsettle the route table cache. In order to combat the incoming stream, it may be useful to pick a handful of sources and cycle through them. This approach has not shown much promise when used within Lucent, perhaps because many Lucent routers use only a single default route so forwarding cache state is not a resource issue.

Results

We obtained logins on various hosts throughout Lucent's intranet. We ran a non-privileged program named `sendudp` to generate a stream of packets back to a nonexistent host our local network. In most cases we could trace the packets back to the "attacking" building. In many we could traceback to the individual Ethernet.

Some links did not respond to our applied load. In some cases we had to go a hop beyond the non-responding links (all the links that are connected to a machine that are one hop away from where we had traced back to) in order to find a link which, when loaded, affected the packet flow. Sometimes, we could pick up enough of a signal from one of these next layer links that we could continue. It was a quite manual process, however, which could become difficult on unstable links with a large number of incident links.

With two exceptions, corporate users appeared to be ignorant of our tests. If the mapping is subtle, and the load applied for short periods, users are unlikely to notice the performance hit, or dismiss it as normal network variability.

Early on, we confined our testing to a few networks, and the network administrator received enough complaints to notice our activities several times. Our subsequent tests appeared to be unnoticed, though in neither case did we attempt to hide our activities.

On one network which we used to generate load, the broadcast UDP chargen packet initiated a broadcast storm on their network. This network had a gain in the tens of thousands. Local users definitely noticed every time we used it, since it brought the network to a halt. Of the 2,000 networks in Lucent, only this one appeared to be unstable in this matter. It is unlikely that our probe packets would be detected on such a poorly-run network, which is likely to have frequent packet storms from other causes. The Internet likely has an even lower rate of misbehaving networks.

Alternative Strategies

This solution is not the only possible one to DoS attacks. Since DoS attacks rely on anonymity, a solution must eliminate some anonymity of hosts. There are two basic methods to do this: ensure that sufficient spoofed packets are never transmitted over the Internet and developing a method for tracing back packets if necessary. The first two methods discussed below attempt to stop some of the spoofing, by ensuring that the at least the source IP address is on the same network as the actual source of the packets. The last three methods discuss alternative methods of tracing packets.

The problem with many of these is that they require universal deployment in order to work. If a couple ISPs opt to not follow the method, then the attacker can just launch the DoS from such a network.

Filter Return Addresses at the Source

There are many ways to solve the problem of anonymous packets. The most desirable is to enforce correct source addresses at or near their source via a method called ingress filtering [10]. A company or university should block outgoing packets that do not have appropriate return addresses. ISPs should have similar filters for each of their customers. Many firewalls do this as a matter of course.

This solution is undoubtedly the right one. Anonymous packets have no place on the Internet. However, these filters do make life more complicated, and for large users behind slow routers they can even degrade performance. For network administrators, these filters are an additional administrative problem: one more thing to install, maintain, and get wrong. Several RFC's recommend it as essential for any responsible participant in the global routing system. Most firewalls have the ability and capacity to perform these checks. The source-based filtering may upset mobile networking methodologies.

Filtering in Backbone Routers

Routers at the core of the Internet, those running BGP4 and exchanging full Internet routing tables, inherently enforce proper destination addresses on packets, since the routing system is built around forwarding the packet toward the value of this field. Theoretically, routers could perform a similar check on the source address, i.e., drop those with source addresses that are inconsistent with their incoming interface.

Unfortunately, the verification is not nearly so simple, since a packet may come from more than one possible incoming interface, so routers would have to maintain a huge amount of state. Not only do routers not have spare memory resources to maintain this state, they do not have spare CPU resources to perform the verification. In the midst of sustained forwarding rates of millions of packets per second, often operating quite near if not at their maximum capacity, router designers must optimize for speed. An additional lookup of the source information would require similar optimization, and subsequent re-engineering of many routers, an expensive and unlikely scenario unless ISPs are willing to pay for it.

One could imagine that legal fallout from a particularly damaging attack might force this scenario, and some routers may emerge that support such functionality service without re-engineering. In general, however, the industry has long resisted source-based policy routing, and we do not expect a fundamental change in this mind-set in the short to medium term.

Tracing by Hand

The obvious ad hoc solution to finding a spoofing host is to trace packets back to their physical source manually. This is done by contacting an ISP and having them test each link to determine if a large number of packets are traversing that link destined for

the victim network. This is done in a tree-like manner similar to ours, or at the access points to their networks. There are two basic methods to do this, either examine the traffic flow across the link, or manually disconnect a link and see if it alters the packet flow (essentially what we attempt to do without physical access).

This method requires significant cooperation and attention from intervening ISPs, which has proven a problem in past incidents. They may not have the policy, inclination, time, expertise, or the instrumentation to help out. Test equipment may not be available for some locations or links within their network. For example, some Cisco routers have been known to crash if IP DEBUG is used under sufficiently heavy load.

Further, the traces may be needed off-hours: the Panix attack started a little after five one Friday afternoon. It may be hard to find someone at any hour at the ISP who can handle the technical details. Sometimes attacks are only solved because a victim happens to be well-connected to admin-able friends at ISPs that are willing to help them out.

This cooperation is very helpful, but selective, and slows the process down immensely. A quicker method would be extremely useful.

Shutting Down a Router

One could imagine sending a message to a router requesting that it drop all packets for a particular destination for a second or so. This interruption would be long enough to detect a break in incoming packets, without noticeably affecting service. Implementing this feature would provide an obvious denial-of-service attack of its own. The router could require that requests be strongly authenticated, but there is no infrastructure present for such validation in the current Internet. In self-defense, a router would have to do a similar rate-limiting as it does with UDP responses, rendering the feature useless for a significant attack. Given the ease that an attacker can hide her attack, it probably is not worth deploying such a service.

Marking Packets with IP Addresses

Another alternative is to place the IP address of all the routers that a packet goes through during its flight across the Internet. This has two obvious disadvantages: it requires CPU time of the routers and it increases the size of packets, especially in the case of routing loops. Both of these could be reduced by having it mark only every 1 in n packets through a given interface. If n is small enough, a long enough attack would give you the complete list of routers along the path, if not their actual order. If n is chosen large enough, the additional router time and packet size increase would be negligible. In practice, one might want to randomly vary N to avoid possible problems with routers synchronizing. If n is too small, than the attacker can insert packets into the network that can "fool" your system into misdiagnosing the path. In

practice, you may want to keep only one address in a packet at a time in order to simplify the header.

Ethics

We acknowledge that our methods to traceback anonymous packets resemble techniques used by hackers. There are several questions to deal with in this area.

1. *Does the tracking attempt cause more damage than the actual packets?* Obviously if the answer is yes, then we should not pursue the technique. We cannot provide a universal answer to this question; it really depends on the situation. If the anonymous packet stream has shut down the daytime service on your web server, it is perhaps not costing you enough to take serious action. If they are crashing your network and denying your customers access to the service you sell them, then perhaps stopping is worth the cost of congesting a few network links for a few seconds (it almost certainly seems so to you). The user of this method will have to make this judgment.
2. *Does leaving a service (such as UDP chargen) enabled on your machine implicitly mean you have given permission to use it?* Our method does not attempt to gain access to private information or crash individual machines, but it does leverage accessible services from private machines. However, these machines have left the UDP chargen enabled (or whatever service is employed).

The easy answer is yes, but it runs dangerously close to the hacker's defense that running a service with possible security holes indemnifies those who intentionally exploit it. On the other hand, we are not really exploiting a security in an implementation. Indeed, we are following the intended protocol specification exactly. Nonetheless, the essence of our tool is the imposition of a denial-of-service of the attacker's own denial-of-service attack against us.

After much consideration, we must conclude that the appropriate answer to this question comes down to motivation. While a hacker is generally trying to harm the machine, gain access to private information, or journey on an ego trip, our tool is leveraging the machine for a secondary purpose that is helpful to the Internet community.

We recognize that this argument may not be sufficient for some organizations that reside on the Internet.

Discussion

This technique is not ideal, either in efficiency, speed, or impact on other Internet users. We have shown that it works on an intranet, which tends to be a more controlled environment than the Internet itself.

It would be preferable to find a better solution involving ISP coordination and cooperation. Unfortunately, we have to admit that sometimes the perpetrators are *at* ISPs, so an official mechanism that tips them off might be completely impotent. We expect that ISPs will drift toward better solutions as their own clients demand assistance.

Acknowledgments

Alexis Rosen and Simona Nass were very helpful in providing information and access during the Panix attack. Peter Winkler and Diane Litman helped us with statistical analysis of perturbed packets. Tom Limoncelli gave helpful information about Lucent's intranet. Andrew Gross, k claffy, Doug Comer, Mike O'Dell, and Marcus Ranum provided a number of useful insights and comments on the issues and techniques raised here.

Author Information

Hal Burch earned his B.S. in Mathematics, Computer Science, and Physics from University of Missouri-Rolla in 1997 and his M.S. in Computer Science from Carnegie Mellon University in 2000. He is now working on his doctorate at Carnegie Mellon while employed by Lumeta Corporation. He is a coach from the U.S.A. Computing Olympiad. Reach him at hburch@cs.cmu.edu or hburch@lumeta.com; see his web page at <http://www.cs.cmu.edu/~hburch>.

Cheswick has worked on (and against) operating system security for nearly 30 years. Starting in 1987, he worked at Bell Laboratories on firewalls, PC viruses, network mapping, and Internet security. He co-authored the first full book on firewalls and Internet security with Steve Bellovin. The Internet maps he has created with Hal Burch have appeared on the cover of *Nature*, in *Wired*, and the *National Geographic*. Ches recently left the Labs in a small spinoff, Lumeta Corp., that is mapping and scanning corporate intranets. In his spare time he launches high-power rockets with his wife, works on exhibits for science museums, and automates his home. Reach him at ches@lumeta.com.

Introduction

- [1] Cheswick, B., Burch, H., and Branigan, S., "Mapping and Visualizing the Internet", to appear in Proceedings of USENIX Annual Technical Conference 2000.
- [2] Postel, J., "RFC 791: Internet Protocol," The Internet Society, Sept 1981.
- [3] Postel, J., "RFC 768: User Datagram Protocol," The Internet Society, Aug 1980.
- [4] Postel, J., "RFC 792: Internet Control Message Protocol," The Internet Society, Sept 1981.
- [5] Postel, J., "RFC 864: Character Generator Protocol," The Internet Society, May 1983.
- [6] Govindan, R. and Tangmunarunkit, H., "Heuristics for Internet Map Discovery," Technical

Report 99-717, Computer Science Department, University of Southern California.

- [7] Claffy, K. "Internet measurement and data analysis: topology, workload, performance and routing statistics," NAE '99 workshop
- [8] CERT, "smurf IP Denial-of-Service Attacks," CERT advisory CA-98.01, Jan, 1998.
- [9] CERT, "Results of the Distributed-Systems Intruder Tools Workshop", The CERT Coordination Center, Dec, 1999.
- [10] Ferguson, P. and Senie, D. "RFC 2267: Network Ingress Filtering: Defeating Denial of Service Attacks which employ IP Source Address Spoofing," The Internet Society, Jan, 1998.
- [11] CERT, "TCP SYN Flooding and IP Spoofing Attacks," CERT Advisory CA-96.21, Sept, 1996.
- [12] CERT, "IP Spoofing Attacks and Hijacked Terminal Connections," CERT Advisory CA-95.01, Jan, 1995.

Analyzing Distributed Denial Of Service Tools: The Shaft Case

Sven Dietrich – NASA Goddard Space Flight Center

Neil Long – Oxford University

David Dittrich – University of Washington

ABSTRACT

In this paper we present an analysis of Shaft, an example of *malware* used in distributed denial of service (DDoS) attacks. This relatively recent occurrence combines well-known denial of service attacks (such as TCP SYN flood, smurf, and UDP flood) with a distributed and coordinated approach to create a powerful program, capable of slowing network communications to a grinding halt.

Denial of service attack programs, *root kits*, and network *sniffers* have been around in the computer underground for a very long time. They have not gained nearly the same level of attention by the general public as did the Morris *Internet Worm* of 1988, but have slowly progressed in their development. As more and more systems have come to be required for business, research, education, the basic functioning of government, and now entertainment and commerce from people's homes, the increasingly large number of vulnerable systems has converged with the development of these tools to create a situation that resulted in distributed denial of service attacks that took down the largest e-commerce and media sites on the Internet.

In contrast, we provide a comparative analysis of several distributed denial of service tools (e.g., Trinoo, TFN, Stacheldraht, and Mstream), look at emerging countermeasures against some of these tools. We look at practical examples of these techniques, provide some examples from test environments and finally talk about future trends of these distributed tools.

Introduction

Network-based attacks are nothing new, but up to last year the techniques utilized were focused on simple point-to-point denial of service. By denial of service we mean overwhelming the victim host or network to the point of unresponsiveness to the legitimate user. We provide a little overview, by no means complete, of previous point-to-point denial of service techniques. There are four major point-to-point techniques: TCP SYN flooding, UDP flooding, ICMP flooding, and Smurf attacks. The first one misbehaves from the standard three-way TCP handshake causing resource consumption and bandwidth consumption, whereas the remaining ones intend to consume the victim's bandwidth.

The year 1999 saw an emergence of new denial of service tools. The change was inevitable: the growth of network pipes made simple point-to-point tools either useless, or the improved tracking capabilities easily shut down the source of the problem. Even though some solutions or at least containment methods exist for the above, the distributed variants as an evolution of coordinated many-to-one attacks escape the traditional model sufficiently. Rather than relying on a single source, attackers could now take advantage of some hundred, thousand, even ten thousand or more systems to inflict denial of service onto their victims.

Analysis

The DDoS Network Model

A Distributed Denial of Service Network follows a hierarchical model, with one or more *attackers* controlling a so-called *handler*, which in turn controls the hordes of *agents* that execute the commands relayed to them.

The communication between the attacker and the handler, and between the handler and the agents is referred to as the *control traffic* of the network, whereas the communication between the agents and the victims is referred to as the *flood traffic*. Control traffic can be TCP, UDP, ICMP, or a combination of the three. Flood traffic consists of traffic generated by each individual point-to-point denial of service technique, or sometimes a combination thereof.

In order to remove himself from view, the attacker introduces additional layers between the victim host(s) and himself. He can access the handler via a variety of mechanisms, the most popular being a simple *telnet*. More sophisticated tools use, or can take advantage of, more advanced techniques, such as encrypted TCP connections (ssh is a possibility for TFN, blowfish-encrypted proprietary as in Stacheldraht) or non-standard methods such as embedding commands in ICMP or UDP packets (e.g., LOKI [22, 23] or Q). Additional care is taken to protect the handler, as it is the key control point and effectively the

anonymizer of the network. So as to eliminate a single point of failure, more than one handler is found in practice, and in most cases each handler has equal power over its agents.

The agents are controlled from the handler, often using a different protocol than the one in effect between attacker and handler. It is speculated that this is done to evade correlation. The communication is not necessarily bidirectional, as there have been cases of oblivious transfers. Instructed by the attacker, the handler can control the numerous agents to perform the attacks by proxy. As we will see, some DDoS tools provide a clear overview of the DDoS network, e.g., enabling to determine the status and performance of each individual agent.

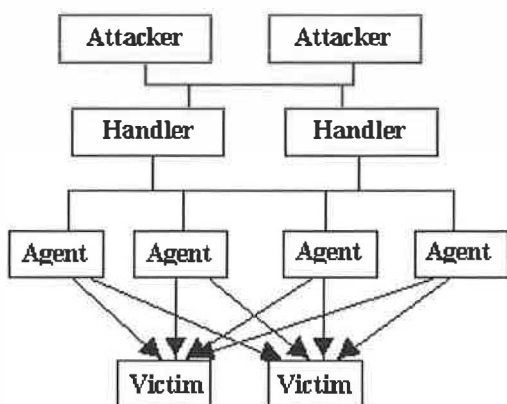


Figure 1: A typical DDoS network

Findings

Shaft was initially detected through anomalous network activity. With the help of the network analyzer Argus [1], spikes (see Figure 2) in the packet flows led to the discovery of the *shaftnode* agent on the compromised system within the local network. It was one of about 100 nodes in a Shaft DDoS network. The successful retrieval of an attack binary, the *shaftnode* agent, and eventually its source, permitted a thorough analysis of its functionality.

Since the *shaftmaster* handler was not retrieved until four months later, it took simulation, thorough analysis of Argus [1] logs and a pinch of creativity to reconstruct the functionality of this attack tool. Simulation and analysis tools such as the Unix debugging tool *strace*, and disassembly of binaries are the main contributors to the understanding of Shaft.

Communication Features

As a first step, it was important to identify the network communication aspect of Shaft. Shaft (in the analyzed version, 1.72) is modeled after Trinoo [11], in that communication between handlers and agents is achieved using the unreliable IP protocol UDP. See Stevens [29] for an extensive discussion of the TCP and UDP protocols. Remote control is established via

a simple telnet connection to the handler. Shaft uses *tickets* for keeping track of the transactions issued to its individual agents. Both passwords and ticket numbers have to match for the agent to execute the request. A simple letter-shifting (Caesar cipher, see Schneier [27]) is in use.

Command Structure

Next, analyzing the command structure of the tool provided additional understanding of the capabilities of Shaft. Using available source and the simple Unix command *strings*, we established the command syntax of both the agent and the handler. It provided insight into the capabilities of the handler that was not apparent from the agent source. A full listing of both the agent and handler commands can be found in Appendix 1 and 2, respectively.

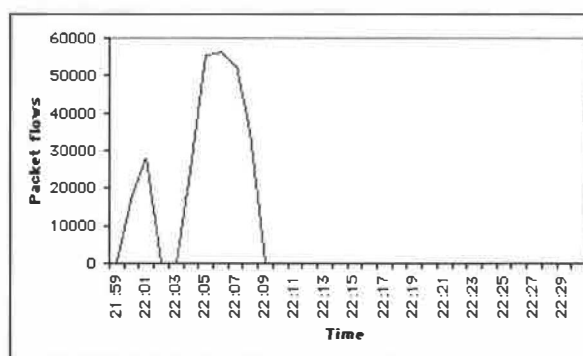


Figure 2: Spikes in network activity.

Detection

Brief Description of Installation Methods

As with previous DDoS tools, the methods used to install the handler/agent will be the same as installing any program on a compromised Unix system, with all the standard options for concealing the programs and files (e.g., use of hidden directories, *root kits*, kernel modules, etc.). The reader is referred to Dittrich's Trinoo analysis [11] for a description of possible installation methods of this type of tool.

Further findings [32] have revealed that the Shaft DDoS tools were indeed used in conjunction with a *root kit*, an *inetd*-based (*inetd* is the Unix server that handles most incoming connections such as telnet and ftp) trojan, a trojaned secure shell (SSH) daemon, and a set of Unix shell scripts to automatically distribute the tools out to the individual agent systems. The present *inetd*-based trojan has been known to exist in the wild as early as May 1999.

The distribution Unix shell script (from [32]), as sent with netcat [19] to the trojaned system, is as follows:

```

#!/bin/sh
echo "oir##t"
echo "QUIT"
sleep 5
echo "cd /tmp"
  
```

```

sleep 5
echo "rcp user@host:shaftnode ./"
sleep 5
echo "chmod +x shaftnode"
sleep 5
echo "./shaftnode"
echo "exit"

```

This shell script installs the shaftnode agent on the system, by performing a remote copy from a repository host into the /tmp directory, making it executable and launching it. The reader is referred to [32] for a complete discussion of the installation, trojaning and rootkit-ing of the handler host.

Algorithmic Overview of Attacks

Upon launch, the shaft agent (the “shaftnode”) reports back to its default handler (its “shaftmaster”) by sending a “new <upshifted password>” command, which registers the new agent in the pool of agents available to the handler. For the default password of “shift” found in the analyzed code, this would be “tijgu”. Therefore a new agent would send out “new tijgu”, and all subsequent messages would carry that password in it. Only in one case does the agent shift in the opposite direction for one particular command, e.g., “pktres rghes”. While it was initially unclear whether this was a mistake, a more thorough analysis

of the shaftmaster revealed that both shifts were used in an attempt to evade analysis.

Incoming commands arrive as space separated items: command, upshifted password, command argument, socket number, ticket, and optional arguments, which can be represented as the message flow diagram between handler H and agent A:

1. $A \rightarrow H: \text{"new"}, f(\text{password})$
 2. $H \rightarrow A: \text{cmd}, f(\text{password}), [\text{args}], Na, Nb$
 3. $A \rightarrow H: \text{cmdrep}, f(\text{password}), Na, Nb, [\text{args}]$
 4. Jump to step 2.
- $f(X)$ is the Caesar cipher function on X
 - Na, Nb are numbers (tickets, socket numbers)
 - $\text{cmd}, \text{cmdrep}$ are commands and command acknowledgments
 - args are command arguments

The flooding occurs in bursts of 100 packets per host, with the source port and source address randomized. This number is hard-coded, but it is believed that more flexibility can be added. Whereas the source port spoofing only works if the agent is running as a root privileged process, the author has added provisions for packet flooding using the UDP protocol and with the correct source address in the case the process is running as a simple user process. It is noteworthy that the random function is not properly seeded, which may

Time	Protocol	Src IP/Port	Flow	Dst IP/Port
21:39:22	tcp	z.z.z.z.53982	↔	x.x.x.x.21
21:39:32	tcp	x.x.x.x.1023	→	y.y.y.y.514
21:39:56	udp	x.x.x.x.33198	→	z.z.z.z.20433
21:45:20	udp	z.z.z.z.1765	→	x.x.x.x.18753
21:45:20	udp	x.x.x.x.33199	→	z.z.z.z.20433
21:45:59	udp	z.z.z.z.1866	→	x.x.x.x.18753
21:45:59	udp	x.x.x.x.33200	→	z.z.z.z.20433
21:45:59	udp	z.z.z.z.1968	→	x.x.x.x.18753
21:45:59	udp	z.z.z.z.1046	→	x.x.x.x.18753
21:45:59	udp	z.z.z.z.1147	→	x.x.x.x.18753
21:45:59	udp	z.z.z.z.1248	→	x.x.x.x.18753
21:45:59	udp	z.z.z.z.1451	→	x.x.x.x.18753
21:46:00	udp	x.x.x.x.33201	→	z.z.z.z.20433
21:46:00	udp	x.x.x.x.33202	→	z.z.z.z.20433
21:46:01	udp	x.x.x.x.33203	→	z.z.z.z.20433
21:48:37	udp	z.z.z.z.1037	→	x.x.x.x.18753
21:48:37	udp	z.z.z.z.1239	→	x.x.x.x.18753
21:48:37	udp	z.z.z.z.1340	→	x.x.x.x.18753
21:48:37	udp	z.z.z.z.1442	→	x.x.x.x.18753
21:48:38	udp	x.x.x.x.33204	→	z.z.z.z.20433
21:48:38	udp	x.x.x.x.33205	→	z.z.z.z.20433
21:48:38	udp	x.x.x.x.33206	→	z.z.z.z.20433
21:48:56	udp	z.z.z.z.1644	→	x.x.x.x.18753
21:48:56	udp	x.x.x.x.33207	→	z.z.z.z.20433
21:49:59	udp	x.x.x.x.33208	→	z.z.z.z.20433
21:50:00	udp	x.x.x.x.33209	→	z.z.z.z.20433
21:50:14	udp	z.z.z.z.1747	→	x.x.x.x.18753
21:50:14	udp	x.x.x.x.33210	→	z.z.z.z.20433

Table 1: Compromise flow on Nov 28.

lead to predictable source port sequences and source host IP sequences.

The source port is generated with $(R \bmod (65535-1024)+1024)$ where R is the output of the `rand()` function. This will generate source ports greater than 1024 at all times.

The source IP is of the form $R1.R2.R3.R4$ where $R1, R2, R3, R4$ are the outputs of $\text{rand()} \bmod 255$. The source IP numbers can (and will) contain a zero in the leading octet.

Additionally, the sequence number for all TCP packets is fixed, namely 0x28374839, which helps with respect to detection at the network level. The ACK and URGENT flags are randomly set, except on some platforms. Destination ports for TCP and UDP packet floods are randomized.

The client must choose the duration ("time"), size of packets, and type of packet flooding directed at the victim hosts. Each set of hosts has its own duration, which gets divided evenly across all hosts. This is unlike TFN [6] which forks an individual process for each victim host. For the type, the client can select UDP, TCP SYN, ICMP packet flooding, or the combination of all three. Even though there is potential for having a different type and packet size for each set of victim hosts, this feature is not exploited in this version.

When a general command is issued, it is sent to all hosts listed in a hidden file containing all the Shaft agents, in general with a timeout of 30 seconds. To date, no mechanism to alter that timeout has been found. Some commands have longer timeouts, up to 300 seconds. A list of outstanding tickets (transactions waiting to complete) is available to the attacker with the "ltic" command, which lists the ticket number and its corresponding remaining time. The attacker can visually correlate the ticket number to the actual command by scrolling back in his screen buffer and comparing the number that was printed after the execution of the command, similar to seeing a process id displayed when sending a process into the background on a Unix system.

The author of Shaft seems to have a particular interest in statistics, namely packet generation rates of its individual agents. The statistics on packet generation rates are possibly used to determine the "yield" of the DDoS network as a whole. This would allow the attacker to stop adding hosts to the attack network when it reached the necessary size to overwhelm the victim network, and to know when it is necessary to add more agents to compensate for loss of agents due to attrition during an attack (as the agent systems are identified and taken off-line).

Packet Flow Analysis

In this section we will look at a practical example of an attack carried out with the Shaft distributed denial of service attack tool, as seen from the attacking network perspective.

The handler is listening on port 20433, and an existing connection on port 20432 is awaiting the commands of the attacker. The packet flow is illustrated in Table 1.

There is quite some activity between the handler and the agent, as they go through the command request and acknowledgement phases. There was also what appeared to be testing of the impact on the local network itself with, among others, UDP packet flooding against the broadcast address (first 2-3 spikes), followed by ICMP flooding as shown in Figure 2. See Figure 3 for a fine-grained view.

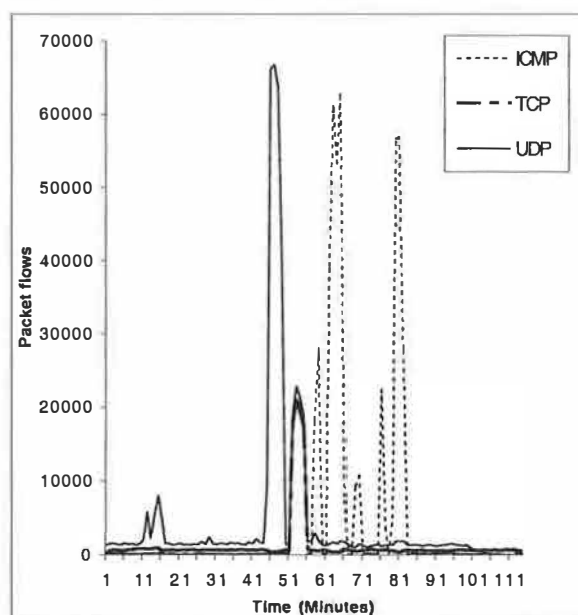


Figure 3: 28 Nov 1999 floods 21:00-23:00.

The interesting portion is the first three lines. It shows the penetration from the handler (z.z.z.z) using the inetd-based trojan with source port 53982 and destination port 21 (any inetd related port would have worked), the download of the shaftnode binary from y.y.y.y via rcp (remote copy, port 514), and the registration of the shaftnode agent with its shaftmaster handler. The theory that these were the traces of the penetration was confirmed by findings [32] on the handler host. The ten second delay between the packet on port 21 and the remote copy on port 514 is consistent with the script mentioned in the section on installation methods. Later that night, the attacker performed several attacks (three UDP and one combination TCP/UDP/ICMP) in order to test the Shaft network further, as illustrated in Figure 4. Let us look at the individual phases from a later attack after it became possible to record the packet contents, as well as general flow data, subsequent to a determination of the agent, handler and communication ports. Subsequently, the handler continued to send such packets even though the agent had been disabled and the host integrity recovered. This is illustrated in Table 2.

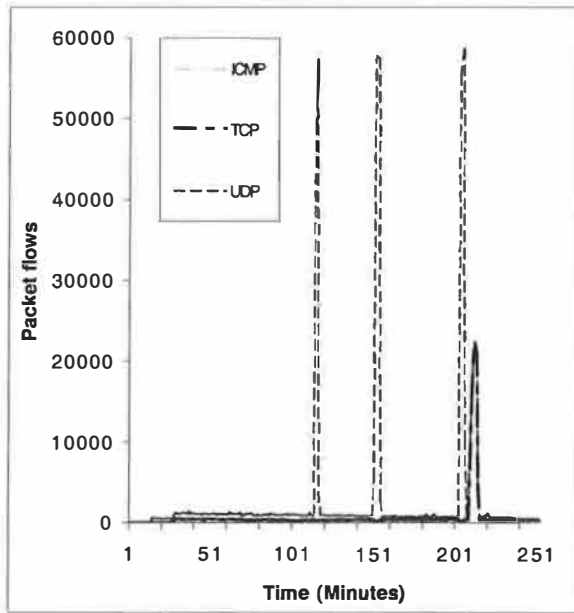


Figure 4: Further testing 29 Nov 1999 02:00-07:00.

time	flow	command
18:06:40	Z → X	alive tijgu hi 5 8170
18:09:14	Z → X	time tijgu 700 5 6437
18:09:14	X → Z	time tijgu 5 6437 700
18:09:16	Z → X	size tijgu 4096 5 8717
18:09:16	X → Z	size tijgu 5 8717 4096
18:09:23	Z → X	type tijgu 2 5 9003

Table 2: Setup and configuration phase on Dec 4.

time	flow	command
18:09:24	Z → X	own tijgu a.a.a.a 5 5256
18:09:24	X → Z	owning tijgu 5 5256 a.a.a.a
18:09:24	Z → X	pktres tijgu a.a.a.a 5 1993
18:09:24	Z → X	own tijgu b.b.b.b 5 78
18:09:24	Z → X	pktres tijgu j.j.j.j 5 8845
18:09:24	Z → X	own tijgu c.c.c.c 5 6247
18:09:25	Z → X	own tijgu d.d.d.d 5 4190
18:09:25	Z → X	own tijgu e.e.e.e 5 2376
18:09:25	X → Z	owning tijgu 5 78 b.b.b.b
18:09:26	X → Z	owning tijgu 5 6247 c.c.c.c
18:09:27	X → Z	owning tijgu 5 4190 d.d.d.d
18:09:28	X → Z	owning tijgu 5 2376 e.e.e.e
18:21:04	X → Z	pktres rghes 5 1993 51600
18:21:04	X → Z	pktres rghes 0 0 51400
18:21:07	X → Z	pktres rghes 0 0 51500
18:21:07	X → Z	pktres rghes 0 0 51400
18:21:07	X → Z	pktres rghes 0 0 51400

Table 3: Host list and statistics.

The handler issues an “alive” command, and says “hi” to its agent, assigning a socket number of “5” and a ticket number of 8170. We will see that this “socket number” will persist throughout this attack. A time period of 700 seconds is assigned to the agent, which is acknowledged. A packet size of 4096 bytes is

specified, which is again confirmed. The last line indicates the type of attack, in this case “the works”, i.e., UDP, TCP SYN and ICMP packet flooding combined. Failure to specify the type would make the agent default to UDP packet flooding.

Now the list of hosts to attack and which ones they want statistics from on completion, as shown in Table 3. To protect the identity of the victims, the hosts IP number have been replaced with a.a.a.a through j.j.j.j.

time	flow	command
18:24:25	Z → X	own tijgu e.e.e.e 5 4493
18:25:53	Z → X	own tijgu b.b.b.b 5 9392
18:27:05	Z → X	own tijgu a.a.a.a 5 3085
18:27:06	X → Z	owning tijgu 5 3085 a.a.a.a
18:33:52	Z → X	own tijgu c.c.c.c 5 1878
18:33:53	X → Z	owning tijgu 5 1878 c.c.c.c
18:36:04	X → Z	pktres rghes 0 0 104100
18:36:20	Z → X	pktres tijgu a.a.a.a 5 1511
18:36:21	X → Z	owning tijgu 5 1754 a.a.a.a
18:37:33	X → Z	pktres rghes 0 0 81700
18:38:13	Z → X	own tijgu f.f.f.f 5 3126
18:38:13	Z → X	pktres tijgu f.f.f.f 5 4697
18:38:14	X → Z	owning tijgu 5 3126 f.f.f.f
18:38:47	X → Z	pktres rghes 5 1511 76600
18:39:15	Z → X	own tijgu g.g.g.g 5 4272
18:39:16	X → Z	owning tijgu 5 4272 g.g.g.g
18:39:41	Z → X	own tijgu c.c.c.c 5 8850
18:39:41	Z → X	pktres tijgu c.c.c.c 5 9924
18:40:43	Z → X	own tijgu c.c.c.c 5 2672
18:41:25	X → Z	owning tijgu 5 5195 h.h.h.h
18:45:33	X → Z	pktres rghes 5 9924 53700
18:48:01	X → Z	pktres rghes 0 0 48800
18:49:54	X → Z	pktres rghes 5 4697 45700
18:50:56	X → Z	pktres rghes 0 0 44900
18:51:22	X → Z	pktres rghes 0 0 45700
18:53:04	X → Z	pktres rghes 0 0 63700
18:54:47	Z → X	own tijgu i.i.i.i 5 2086
18:54:47	Z → X	pktres tijgu i.i.i.i 5 6980
18:54:47	X → Z	owning tijgu 5 2086 i.i.i.i
19:06:27	X → Z	pktres rghes 5 6980 241200

Table 4: More hosts and statistics.

Now that all other parameters are set, the handler issues several “own” commands, in effect specifying the victim hosts. Those commands are acknowledged by the agent with an “owning” reply. The flooding occurs as soon as the first victim host gets added. The handler also requests packet statistics from the agents for certain victim hosts (e.g., “pktres tijgu a.a.a.a 5 1993”). Note that the reply comes back with the same identifiers (“5 1993”) at the end of the 700 second packet flood, indicating that 51600 sets of packets were sent. One should realize that, if successful, this means 51600 x 3 packets due to the configuration of all three (UDP, TCP, and ICMP) types of packets. In turn, this results in roughly 220 4096 byte packets per second per host, or about 900 kilobytes per second per

victim host from this agent alone, about 4.5 megabytes per second total for this little exercise. A graphical view can be seen in the first portion (minutes 1 through 12) of Figure 5.

Note the reverse shift ("shift" becomes "rghes", rather than "tijgu") for the password on the packet statistics. Continuing on with the attack, as shown in Table 4, the attacker selects new targets in a staggered manner, but still keeping the established settings of the 700 second combination type attack of 4096 byte packets. The yields of the attack vary from roughly 800 kilobytes per second per host in a multi-target setting to 4.2 megabytes per second per host in a single target setting (Target I). The staggered approach can be observed in the right two thirds of Figure 5 (minutes 14 through 50).

Cryptographic Aspects

Shaft incorporated several noteworthy techniques for keeping information secret. For one, the letter-shifting or Caesar cipher, was applied several times within this tool. As described previously, the transaction password "shift" was shifted by one letter upwards to generate the string "tijgu" observed on the network. However, a different shift in the opposite direction generated "rghes" for the return statistics. While the author(s) of this program did not encrypt the entire message exchange between handler and agent, they did nevertheless *obfuscate* the real strings, such as applying the shift to the handler IP numbers in the binary and also the port numbers in the case of a "switch" command, namely by adding an offset to the real port number.

As with the original Trinoo tool, the Shaft handler contained 13-character strings, strangely resembling Unix crypt() output. Through close analysis of the handler code, it was established that they represented the access passwords to the control port of the program, that is where the attacker would connect to and perform the distributed denial of service from a convenient, but not quite menu-driven, command line. The actual passwords were recovered in a similar fashion to the ones from Trinoo and Stacheldraht, except that the above shifts had to be performed on the ciphertext first.

Similar to the handler settings in earlier tools, the author of Shaft attempts to keep the list of its agents in a non-trivial format. Other tools encrypt that list using Blowfish, but this tool packs the four octets of the IP number into a 4-byte integer and writes the ASCII representation of that number to a file, one per line. For example, adding the agent with IP address 127.0.0.1 using "+node 127.0.0.1" would yield a line containing "16777343", which is:

$$127 \times 256^0 + 0 \times 256^1 + 0 \times 256^2 + 1 \times 256^3.$$

In order to extract the IP numbers from the list, one would apply the reverse transformation.

Anomaly Detection

The network flooding which took place was initially noticed after a cursory glance at the hourly network flow data files recorded using an Argus [1] monitor at the main Internet connection point. Without any such monitoring the activities would have almost certainly gone unnoticed since the floods took place over the Sunday-Monday night. Other IDS records

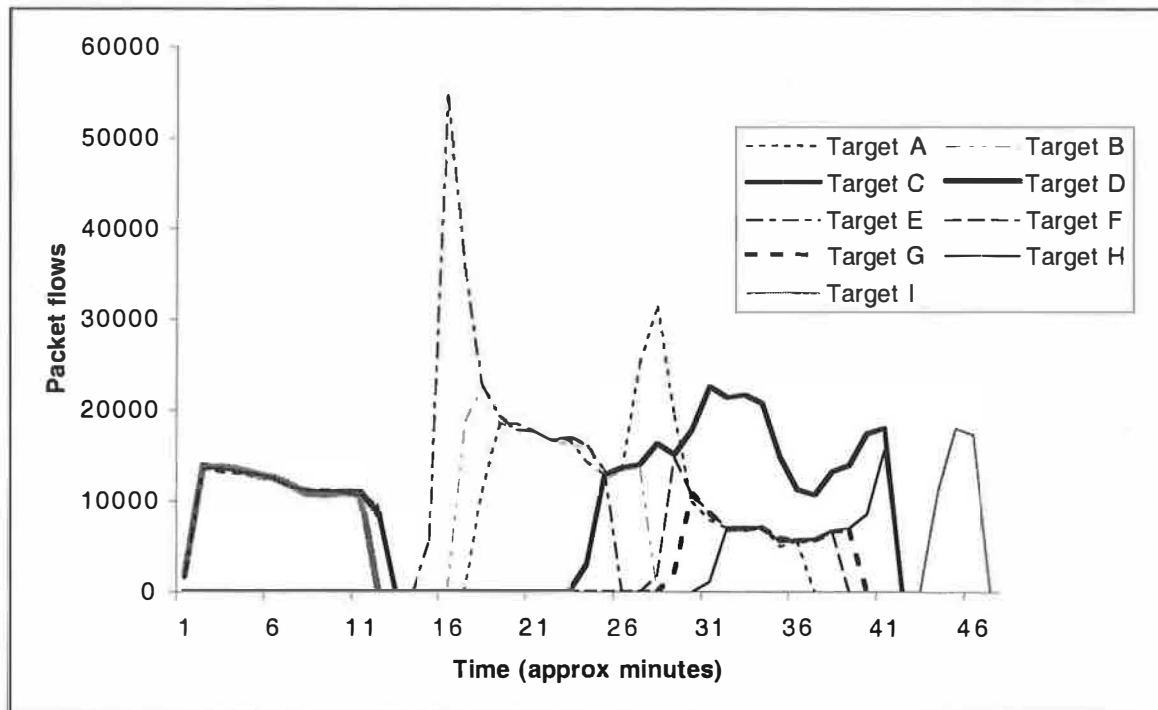


Figure 5: 4 Dec 1999 floods.

indicated a possible UDP portscan had taken place but for host IPs not possible on the local net blocks.

For example, the typical argus data file at that time of day (night) would be 4Mbytes but these grew to between 40 and 100Mbytes. Analyzing such large data files takes significant effort and resources (hence the hourly rotation) but it was possible to determine the start time of the rapid rise in connections and then tracing the external connection (handler) and internal flooder (agent) becomes a matter for trial and error and some measure of good fortune. In this instance the first guess was used to contact the host administrator who was able to locate the process still left running (although inactive), obtain an lsof [18] output and recover residual files and logs.

Reducing the data for the hourly flows down to something suitable for graphical display was complicated by the very large number of data points which overwhelms most of the standard graphing or statistical packages.

Other traffic monitoring applications which might have indicated that an unusually large network flow had occurred would not usually have an accurate time nor could have been used to trace the external → internal communications channel correspondence. For example snmp monitoring of packet numbers is a popular method. Accumulated byte counts per sub-net (host, port, etc) could also have been obtained using NeTraMet [5] but again this would have been inadequate for the post event analysis.

Impact of Victim Hosts/networks

The effect of the combined outpouring of packets has already been considered from the point of view of the target victim but it should be noted that very little legitimate traffic was able to move over the Internet gateway while the flood took place. This secondary denial of service would be of major significance during normal working hours and when combined with several such agents distributed around one site can lead to saturation of the essential backbone infrastructure and routers.

Impact on network – given the time of day it had little or no impact apart from slowing external port scanners (!) – maybe it is worth noting that on typical asynchronous ATM external connections there would be an impact on outgoing versus incoming. The impact of one host running flat out will be a lot less than several hosts running as agents. Deliberately limiting agents to one per site would have considerable benefits when it comes to avoiding detections while still retaining effective DoS of the target(s).

Secondary Effects

Poor DNS response (if any) – even problems managing network devices during the peaks. The flurry of “response” packets caused by the floods can create additional complications within the network. Due to the “inband-signalling” nature of TCP/IP, the

control messages related to network management must travel over the very same congested network.

NIDS vs. Active Scanning

Network Intrusion Detection Systems (NIDS)

During very high (near saturation) flows almost no event of any kind would be logged by an IDS system – they would either have to drop packets at a very high rate or require multi-CPU architectures in order to combine packet collection and packet state analysis. As pipe capacities continue to grow (Gigabit, etc) there will be serious difficulties for network flow monitors such as Argus to keep up based on the typical PC architecture (there is seldom a budget available for a top end machine which will, hopefully, be wasting cycles 99 % of the time waiting for such events).

Passive Scanning

For the purpose of detecting malicious activity, certain features of the whole DDoS package have to be considered and provide clues for passive scanning of such events. This program does not provide for code updates (like TFN or Stacheldraht). This may imply “rcp” or “ftp” connections during the initial intrusion phase (see also [11]). As found in [32], the intruders used “rcp” in their distribution scripts, but this could easily be altered.

The program uses UDP traffic for its communication between the handlers and the agents. Considering that the traffic is not encrypted, it can easily be detected based on certain keywords. Performing an “ngrep” [20] for the keywords mentioned in the syntax sections (Appendix 1 and 2), will locate the control traffic, and looking for TCP packets with sequence numbers of 0x28374839 (decimal 674711609) may locate the TCP SYN packet flood traffic. The latter traffic can be detected through its secondary effect of causing SYN|ACK and RST|ACK traffic with sequence numbers of 0x2837483a (decimal 674711610), as pointed out by Richard Bejtlich (who has been witnessing these effects – with this same sequence number – for well over a year [3]). Source ports of the flood traffic are always above 1024, and source IP numbers can include zeroes in the leading octet.

Strings in this control traffic can be detected with the “ngrep” program using the same technique shown in [11, 12, 13]. Here are some examples that will locate the control traffic between the handler and the agent, independently of the port number used.

```
# ngrep -i -x "alive tijgu" udp
U 192.168.10.1:4001 -> 192.168.10.2:18753
61 6c 69 76 65 20 74 69      alive ti
6a 67 75 20 68 69 20 35      jgu hi 5
20 38 36 34 31 0a           8641.

U 192.168.0.2:1494 -> 192.168.0.1:20433
61 6c 69 76 65 20 74 69      alive ti
6a 67 75 20 35 20 38 36      jgu 5 86
34 31 20 62 6c 61 68        41 blah
```

The above will show the “alive” messages exchanged between handler and agents.

```
# ngrep -i -x "pktres|pktstat" udp
U 192.168.10.2:1499 -> 192.168.10.1:20433
70 6b 74 73 74 61 74 20      pktstat
74 69 6a 67 75 20 35 20      tijgu 5
31 32 35 37 20 30            1257 0
```

The above shows the request for packet statistics and the flood results.

```
# ngrep -i -x "switch tijgu" udp
U 192.168.10.1:4001 -> 192.168.10.2:18753
73 77 69 74 63 68 20 74      switch t
69 6a 67 75 20 32 30 34      ijgu 204
38 33 20 35 20 32 39 36      83 5 296

U 192.168.10.2:1522 -> 192.168.10.1:20433
73 77 69 74 63 68 65 64      switched
20 74 69 6a 67 75 20 35      tijgu 5
20 32 39 36                    296
```

This previous example shows the directive from the handler to “switch” to this handler.

For specific signature detection, one could also use Snort [25]. See the caveats below.

Active Scanning

Scanning the network for open port 20432 will reveal the presence of a handler on your local area network.

For detecting idle agents, one could write a program similar to George Weaver’s trinoo detector. Sending out “alive” messages with the default password to all nodes on a network on the default UDP port 18753 will generate traffic back to the detector, making the agent believe the detector is a handler.

There are also two excellent scanners for detecting DDoS agents on the network: Dittrich’s “dds” [15] and Brumley’s “rid” [6].

“dds” was written to provide a more portable and less dependent means of scanning for various DDoS tools. (Many people encountered problems with Perl and the Net::RawIP library [24] on their systems, which prevented them from using the scripts provided in [11, 12, 13].) Due to time constraints during coding, “dds” does not have the flexibility necessary to specify arbitrary protocols, ports, and payloads. One would need to modify the source slightly to detect shaft agents or handlers. A modified version of “dds”, geared towards detecting only “Shaft” agents, is available [9, 16].

A better means of detecting shaft handlers and agents would be to use a program like “rid”, which uses a more flexible configuration file mechanism to define ports, protocols, and payloads.

A sample configuration for “rid” to detect the Shaft control traffic as described:

```
start shaft
  send udp dport=18753
    data="alive tijgu hi 5 1984"
```

```
recv udp sport=20433
  data="alive" nmatch=1
```

end shaft

Caveats

It should be emphasized again and again that the passive and active detection triggers rely on “old” numbers, strings, etc. and that they are often trivial to modify. Selection and use of such tools and commercial NIDS (in particular) should bear this in mind along with their flexibility to insert the “latest” trigger information that may be provided by security teams and organizations.

Related work

We present a brief overview, in chronological order to the best of our knowledge, of DDoS tools that have been mentioned publically.

Early tools

The early tools appeared in early summer 1998. They were clumsy attempts to naturally evolve beyond coordinated attacks [17], but nevertheless laid the foundation to the subsequent tools. The first of them, fapi, featured UDP, TCP (SYN and ACK), and ICMP Echo floods. Its handler to agent communication was UDP-based. It did not provide easy controls for setting up the DDoS network, and did not handle networks over 10 hosts very well. The second one, fuck_them, was a distributed ICMP Echo Reply flooder, where the attacker either supplied the source address to spoof or randomized source addresses were generated (all 32 bits of the IP address).

Trinoo and variants

Trinoo surfaced in the early summer 1999. It has been extensively scrutinized, and we refer to [11] for a thorough analysis. The tool is capable of only generating UDP packet floods without source address forgery, but has full control features. It was capable of crippling the network of the University of Minnesota [10, 11] for three days, leading to a workshop on the subject [7]. Trinoo has mutated at least twice over the last year.

TFN and variants

TFN, a.k.a. Tribe Flood Network, was introduced in late summer 1999. With its limited control features, it still provided UDP packet flood attacks (it gave homage to Trinoo by calling it “trinoo emulation”), TCP SYN flood attacks, ICMP Echo flood attacks, and Smurf attacks in a distributed fashion. It is capable of spoofing either all 32 bits of the IP source address, or just the last 8 bits. As with Trinoo, this tool has been analyzed thoroughly [12].

TFN2K, or TFN2000, is a further development effort on the basis of TFN. It provides the same attacks as TFN, but can randomly do them all at once. Encryption of the control traffic was added to improve the security of the DDoS network and evade signature detection. Control traffic uses a superposition of UDP,

TCP and ICMP, using oblivious transfers, i.e., the receipt is not acknowledged. For a brief review, see [2].

Stacheldraht and Variants

Stacheldraht, German for “barbed wire,” apparently evolved out of Trinoo and TFN. Analyzed in [13], it has full control features, the same basic attacks and source address forgery as TFN, and as a twist, a Blowfish-encrypted control channel for the attacker. Mutated into StacheldrahtV4 in early 2000, it further mutated into Stacheldraht v1.666, which adds TCP ACK and TCP NUL packet flood attacks, and preconfigured Smurf attacks.

Mstream

As the name suggests, Mstream is a “multiple stream” tool, in reference to the very efficient point-to-point stream TCP ACK flooding tool. It has very limited control features and randomizes all 32 bits of the source IP address. For a review of this tool that appeared in the spring of 2000, please see [14].

Omega

Omega, which appeared in early summer 2000, features TCP ACK packet flooding, UDP packet flooding, ICMP flooding, IGMP packet flooding, and a mix of all four floods. Similar to Shaft, it provides statistics on the floods it produces. It randomizes all 32 bits of the source IP address, and introduces a chat function for communication between attackers.

Trinity and Derivatives

Trinity, and its closely related mutation Entitee, take a new approach on the DDoS model. Rather than relying on a handler network, it takes advantage of an existing Internet Relay Chat (IRC) network for its handler-to-agent communications, making a channel on IRC the “handler.” Besides the up to now well-known UDP, TCP SYN, TCP ACK, TCP NUL packet floods, it introduces TCP fragment floods, TCP RST packet floods, TCP random flag packet floods, and TCP established floods, while randomizing all 32 bits of the source IP address.

myServer

In contrast to the sophistication of Trinity, yet released around the same time in summer 2000, myServer is a simplistic DDoS tool. It relies on external programs to provide the denial of service.

Plague

As a third tool in the same generation as Trinity and myServer, it has become obvious Plague was designed by attackers who are reading these reviews and incorporating new improvements based on them. This tool provides TCP ACK and TCP SYN flooding, with what are claimed as fixes over previous TCP ACK flooding tools.

Defenses and Countermeasures

There is no simple solution that would offer one hundred percent protection against these types of

tools. There are, however, a number of steps that can be taken to minimize the impact [16]. Several proposed schemes are emerging for adding traceability to TCP/IP packets.

What is necessary to defend? One needs to defend the hosts, the local net, and the backbone infrastructure. As per the recommendations in [7], certain ingress and egress filtering can minimize the impact of denial of service attacks that use spoofed IP source addresses by eliminating illegitimate IP source or destination addresses. In practicality, this will not reduce the impact of DDoS tools that do not spoof their address or only spoof the last 8 bits of the IP address, making it appear to be originating from the local network that the agent resides on.

One also tries to track floods and identify their source in order to shut them down one way or another, or minimize the impact. In general, identifying the source of a spoofed IP address requires the collusion of the intermediate hosts in the path between the actual source and the victim suffering from the denial of service attack. Bellovin’s ICMP traceback message scheme [4] addresses that problem by forwarding a signed copy of the transient packet traversing the router in a probabilistic manner. In the proposed version, one in every 20000 packets triggers such behavior, in order to avoid an additional denial of service. Savage et al. take a different approach [26] in inserting partial network path markings into the packet traversing the router in a probabilistic fashion, rather than creating an entirely new packet. Their Fragment Marking Scheme, as pointed out by Song [28], lacks the scalability of dealing with large DDoS networks, causing a large number of false positives. Song’s marking schemes provide more efficient traceback under large scale (say 1500 agents) attacks. Stone suggests an IP overlay network to achieve the tracking and forwarding of interesting packets in his Center-Track [30] scheme.

As mentioned above, anomaly detection can pinpoint the presence of a flood in the first place. Signature detection can either locate known control traffic (either attacker to handler, or handler to agent) or responses from victims.

Future Trends and Evolution

Ever since the introduction of Trinoo, at least eight new tools with varying degrees of sophistication and aimed at creating distributed denial of service have been discovered in a time span less than one calendar year. It is difficult to predict the trends of these tools without ending up being the trendsetter or sparking a new idea for the attacker. These tools have destructive potential and one should remain cautious as to the future directions. It is safe to assume, however, that the trends described in the section on related work will continue, namely in creating distributed variants of existing point-to-point tools.

Conclusion

“Shaft” is another DDoS variant with independent origins. The code recovered did appear to be still in development. Several key features indicate evolutionary trends as the genre develops. Of significance is the priority placed on packet generation statistics which would allow host selection to be refined. The analysis of the code and binary was greatly enhanced by the capture of attack preparation and command packets. The captured packets made it possible to assess the impact of a single agent that managed to saturate the network pipe.

The version analyzed had hooks which would allow for dynamic changes to the master host and control port but not the agent control port. However such items are trivially incorporated and must not be taken to be indicative of any current versions which may be in active use. The obfuscation of master IP, ports and passwords used a relatively simple form of encryption but this could easily be strengthened. Evolutionary findings confirm that information flows back to the authors and cause incorporation of counter-countermeasures as the spiral continues.

The detection of DDoS installations will become very much more difficult as such metamorphosis techniques progress, the presence of such agents will still be more readily determined by analysis of traffic anomalies with a consequent pressure on time and resources for site administrators and security teams.

Acknowledgements

We would like to thank the anonymous referees for their valuable feedback, the CERT Coordination Center for fostering cooperation on DDoS, and also the “last of the quartet,” who wishes to remain anonymous, for participating in numerous discussions on DDoS.

Author Information

Sven Dietrich is a senior security architect for Raytheon ITSS at the NASA Goddard Space Flight Center. His focus is computer security, intrusion detection, the building of a PKI for NASA, and the security of IP communications in space. He can be reached at spock@netsec.gsfc.nasa.gov.

Neil Long is a senior systems administrator at the University of Oxford with an academic background in the physical sciences. He has become increasingly interested in network and computer security matters since the early 1990's. He can be reached at neil.long@computing-services.oxford.ac.uk.

Dave Dittrich is a senior security engineer at the University of Washington, supporting Unix workstation administrators on campus for over ten years. His credits include technical analyses (alone or in teams) of the Trinoo, Tribe Flood Network, Stacheldraht, shaft, and mstream distributed denial of service

(DDoS) attack tools. He can be reached at dittrich@cac.washington.edu.

References

- [1] Argus, <ftp://ftp.sei.cmu.edu/pub/argus/>.
- [2] Barlow, Jason and Woody Thrower, *TFN2K – An Analysis*, http://www2.axent.com/swat/News/TFN2k_Analysis.htm.
- [3] Bejtlich, Richard, Public communication, <http://www.sans.org/y2k/032900.htm>.
- [4] Bellovin, Steve, *ICMP Traceback Messages*, <http://www.research.att.com/~smb/papers/draft-bellovin-trace-00.txt>.
- [5] Brownlee, Nevil, *NeTraMet, A Network Traffic Accounting Meter*, <http://www.auckland.ac.nz/net/NeTraMet/>.
- [6] Brumley, David, *Remote Intrusion Detector*, <http://theorygroup.com/Software/RID>.
- [7] CERT Distributed System Intruder Tools Workshop report, http://www.cert.org/reports/dsit_workshop.pdf, Pittsburgh, PA, December 1999.
- [8] CERT Advisory CA-99-17 Denial-of-Service Tools, <http://www.cert.org/advisories/CA-99-17-denial-of-service-tools.html>.
- [9] Dietrich, Sven, *Distributed Denial of Service Page*, <http://netsec.gsfc.nasa.gov/~spock/ddos.html>.
- [10] Dietrich, Sven, “Scalpel, Gauze, and Decompilers: Dissecting Denial of Service (DDoS),” *USENIX ;login: Magazine*, Special Issue on Security, November 2000.
- [11] Dittrich, David, *The DoS Project's “trinoo” distributed denial of service attack tool*, <http://staff.washington.edu/dittrich/misc/trinoo.analysis>.
- [12] Dittrich, David, *The “Tribe Flood Network” distributed denial of service attack tool*, <http://staff.washington.edu/dittrich/misc/tfn.analysis>.
- [13] Dittrich, David, *The “Stacheldraht” distributed denial of service attack tool*, <http://staff.washington.edu/dittrich/misc/stacheldraht.analysis>.
- [14] Dittrich, David, George Weaver, Sven Dietrich, and Neil Long, *The “mstream” distributed denial of service attack tool*, <http://staff.washington.edu/dittrich/misc/mstream.analysis.txt>.
- [15] Dittrich, David, Marcus Ranum, George Weaver, David Brumley, et al., <http://staff.washington.edu/dittrich/dds>.
- [16] Dittrich, David, *Distributed Denial of Service (DDoS) Attacks/Tools Page*, <http://staff.washington.edu/dittrich/misc/ddos/>.
- [17] Green, John, David Marchette, Stephen Northcutt, Bill Ralph, “Analysis Techniques for Detecting Coordinated Attacks and Probes,” *Proceedings of USENIX Workshop on Intrusion Detection and Network Monitoring*, Santa Clara, California, April 9-12, 1999.
- [18] *Isaf*, <http://vic.cc.purdue.edu/>.
- [19] *netcat*, <http://www.l0pht.com/~weld/netcat/>.
- [20] *ngrep*, <http://www.packetfactory.net/Projects/ngrep/>.

- [21] *Packet Storm Security, Distributed denial of service attack tools*, <http://packetstorm.securify.com/distributed/>.
- [22] *Phrack Magazine*, Volume Seven, Issue Forty-Nine, File 06 of 16, [Project Loki], <http://www.phrack.com/search.phtml?view&article=p49-6>.
- [23] *Phrack Magazine*, Volume 7, Issue 51 September 01, 1997, article 06 of 17, [L O K I 2 (the implementation)], <http://www.phrack.com/search.phtml?view&article=p51-6>.
- [24] *Net::RawIP*., <http://quake.skif.net/RawIP>.
- [25] Roesch, Martin, "Snort – Lightweight Intrusion Detection for Networks," *Proceedings of USENIX LISA 1999*, Seattle, Washington, December 1999.
- [26] Savage, Stefan, David Wetherall, Anna Karlin, and Tom Anderson, "Practical network support for IP traceback," *Proceedings of the 2000 ACM SIGCOMM Conference*, <http://www.cs.washington.edu/homes/savage/papers/Sigcomm00.pdf>. August 2000.
- [27] Schneier, Bruce, *Applied Cryptography, 2nd edition*, Wiley.
- [28] Song, Dawn, and Adrian Perrig, *Advanced and Authenticated Marking Schemes for IP traceback*, Report No. UCB/CSD-00-1107, University of California, Berkeley, June 2000.
- [29] Stevens, W. Richard and Gary R. Wright, *TCP/IP Illustrated, Vol. I, II, and III*, Addison-Wesley.
- [30] Stone, Robert, "CenterTrack: An IP-Overlay Network for Tracking DoS floods," *Proceedings to the 9th USENIX Security Symposium*, Denver, Colorado, August 2000.
- [31] *tcpdump*, <http://www.tcpdump.org/>.
- [32] Wash, Richard and Jose Nazario, *Analysis of a Shaft Node and Master*, http://biocserver.cwru.edu/~jose/shaft_analysis/node-analysis.txt.
- [33] Zuckerman, M. J., "Net hackers develop destructive new tools", *USA Today*, <http://www.usatoday.com/life/cyber/tech/review/crg681.htm>, 7 December 1999.

Appendix 1: Agent Commands

Accepted by agent and replies generated back to the handler:

- size** <size> Size of the flood packets. Generates a "size" reply.
- type** <0|1|2|3> Type of DoS to run 0 UDP, 1 TCP, 2 UDP/TCP/ICMP, 3 ICMP. Generates a "type" reply.
- time** <length> Length of DoS in seconds. Generates a "time" reply.
- own** <victim> Add victim to list of hosts to perform denial of service on. Generates a "owning" reply.
- end** <victim> Removes victim from list of hosts (see "own" above). Generates a "done" reply.

- stat** Requests packet statistics from agent. Generates a "pktstat" reply.
- alive** Are you alive? Generates a "alive blah" reply.
- switch** <handler> <port> Switch the agent to a new handler and handler port. Generates a "switching" reply.
- pktres** <host> Request packet results for that host at the end of the flood. Generates a "pktres" reply.
- Sent by agent:
- new** <password> Registering with the handler
- pktres** <password> <sock> <ticket> <packets sent> Packets sent to the host identified by <ticket> number.

Appendix 2: Handler commands

This is an overview of the command structure:

- mdos** <host list> Start a distributed denial of service attack (mdos = massive denial of service?) directed at <host list>. Sends out "own host" and "pktres" messages to all agents.
- edos** <host list> End the above attack on <host list>. Sends out "end host" messages to all agents.
- time** <length> Set the duration of the attack. Sends out "time <length>" to all agents.
- size** <packet size> Set the packet size for the attack (8K maximum as seen in source). Sends out "size <packet size>" to all agents.
- type** <UDP|TCP|ICMP|ALL> Set the type of attack, UDP packet flooding, TCP SYN packet flooding, ICMP packet flooding, or all three. Sends "type <type>" to all agents.
- +node** <host list> Add new agents.
- node** <host list> Remove agents from pool.
- ns** <host list> Perform a DNS lookup on <host list>.
- lnod** List all agents.
- ltic** List all pending tickets (transactions).
- pkstat** Show total packet statistics for agents. Sends out "stat" request to all agents.
- alive** Send an "alive" to all agents. A possible argument to alive is "hi"
- stat** show status values (length, type of DoS, packet size).
- switch** become the handler for agents. Send "switch" to all agents.
- ver** show version.
- whoami** returns "God".
- exit** self-explanatory.

YASSP! A Tool for Improving Solaris Security

Jean Chouanard – Xerox – Palo Alto Research Center

ABSTRACT

This paper presents YASSP, Yet Another Secure Solaris Package, a set of tools developed to help a systems administrator to secure a Solaris host. It explains the internal mechanism used by YASSP to implement these security changes so that a systems administrator can either use this tool in its current form and localize it, or modify the package source to match his needs.

YASSP is composed of the SECclean package and a set of optional packages providing common useful security related tools. The SECclean internal mechanism used to modify the existing operating system is implemented through the Solaris package mechanism and provides a full un-installation procedure. It has required the development of a specific library of shell functions, to ease file manipulations.

The YASSP project provides to the community an easy path to secure a Solaris host. It has taught us a lot about Solaris internals and package manipulation. YASSP is still young, and ready to be enhanced.

Introduction

The project YASSP was started back in 1997 to fulfill the need for installing exposed Solaris servers. For such servers, Sun's Solaris default installation has the following weakness:

- As standardly installed, Solaris is running many network services. This violates the principle that a machine should provide only those services that its owner intends to provide.
- As standardly installed, all network services are available to any host that requests them, violating the principle of "offer services only to those you intend to use them"
- As standardly installed, system logging is inconsistent.
- As standardly installed, the system does not issue security-warning banner messages to people who attempt to use it.

YASSP's current version was created at Xerox Palo Alto Research Center, as a configurable tool to secure a Solaris host. A more complete history of the project can be found in later in this paper.

YASSP can be used on Solaris 2.6, 7 or 8, sparc or Intel architecture to enhance the security of an end-user workstation or a server.

Philosophy

Before looking at YASSP internals, let us understand the concepts used to build YASSP on. These concepts had driven YASSP implementation, had created their own problems and provided solutions, but have stayed constant over the existence of YASSP which has made it very stable.

- It must play by Sun's rules and not fight Sun: we follow as close as we can Sun standard and rules.

- YASSP tries to be as closely integrated to Solaris as it can. It was built and distributed as a Solaris package. YASSP will follow package rules and Sun standards when they exist. When standard does not exist, YASSP will try to accommodate the different choices the systems administrator may have made.
- It can be installed and un-installed without destruction and restores the same context as before. Particular attention was given to these points, to be sure that the systems administrator will never lose some files or modifications done after YASSP installation to any YASSP configuration files. The goal here is double: be sure that any changes done by YASSP can be undone, but also be sure that any changes done after will not be lost in case of un-installation. That is true for all components of YASSP.
- It should run on a minimal installation, like the core choice of the Solaris installation, or any other.
This is one of the most important, and the most limiting rules, as it directly drove choices we made in the tools and language used in YASSP implementation. Except for a few binaries, the core of YASSP is written in Bourne shell, uses sed and [n]awk.
- Additional scripts, not part of YASSP but referred to as examples on the web documentation may use tools not included in the core install, like Perl.
- Secure and closed by default...
- ... but can be opened up after the installation or modified to be localized.
- YASSP default behavior is to be secure. For each configuration choice YASSP offers, it will by default install the one the YASSP team of developers had consider the most secure.

- Most of these configuration choices are easily reviewed and modified after YASSP install.
- Using a three-layer implementation: from the security issue we are trying to solve, we generate a list of modifications that need to happen; this list is implemented by the package installation (File replaced, file edited, file created...)

Being built as a Sun package, and following Sun rules, this tool cares about the Solaris package database, corrects it when it is wrong, and keeps it accurate over time.

Result of YASSP Installation

After installing SECclean, most of your network services will be turned off and most of the processes started by default under Solaris will not run anymore. Your server will be hardened, perhaps too much. That is the SECclean default: if you do not modify its configuration, or if its configuration does not exist, it will be closed and secure. That fits pretty well for installing an exposed server, but not an end user workstation.

SECclean modification to your system can be summarized as:

- Deleting unwanted files (/etc/auto_home, /etc/auto_master, /etc/dfs/dfstab, /var/spool/cron/crontabs/adm, /var/spool/cron/crontabs/lp, and /var/spool/cron/crontabs/uucp).
- Controlling most of the startup scripts through /etc/yassp.conf.
- Changing default environment setting to make your system more secure (umask, password length, PATH) and make some of these variables available through /etc/yassp.conf (/etc/profile, /etc/default/login, /etc/default/su, /etc/default/passwd, /etc/skel/local.cshrc, /etc/skel/local.profile, /etc/.login, /etc/rmmount.conf) and setting default umask for startup script (/etc/init.d/umask.sh).
- Enabling banner files (/etc/motd, /etc/issue, /etc/ftp-banner).
- Controlling access to the system and to specific commands (/etc/pam.conf, /etc/default/login, /rhosts, /etc/hosts.equiv, /etc/cron.d/at.allow, /etc/cron.d/cron.allow, /etc/default/sys-suspend, /etc/ftpusers, /usr/dt/config/Xaccess, and /etc/dt/config/Xaccess).
- Simplifying network startup, but keeping backward compatibility through /etc/yassp.conf (/etc/init.d/inetsvc, /etc/init.d/inetinit, and /etc/init.d/network).
- More control on RPC services (keyerv and rpcbind with TCP wrapper) [15] (/etc/init.d/rpc).
- Turning off available network services (/etc/inet/inetd.conf).
- System parameters tuning, attempting to prevent and log stack-smashing attacks (/etc/system).

- Network stack performances and security tuning [16] (/etc/init.d/nettune and /etc/default/inetinit).
- Adds few common services definitions (/etc/inet/services).
- Establishes standard syslog configuration (/etc/syslog.conf).
- Miscellaneous and logging (/etc/shells, /var/adm/loginlog, /etc/norouter, and /etc/inittab).
- Package database correction and filemode changes to make them more secure (clean-up and fix-modes).

A more detailed and up-to-date explanation of what is actually done can be found on the web, at: <http://www.yassp.org/internal.html>.

On top of SECclean, YASSP will provide integrity check (tripwire) [4], TCP and RPC controls (tcpd and rpcbind with tcp_wrappers) [15], encrypted channel (ssh) and others log rotation and version control of system files (PARCdaily, GNUrcs, and GNUgzip).

An example of how a system looks after the reboot can be found on Appendix 1.

First, as Sun packages have an important place in YASSP implementation, let us explain what they are and how they work.

Sun Packages Overview

In the Solaris world, application software or system components are delivered in units called packages¹. A package is a collection of files and directories required for implementing a new function, and is usually designed and built by the developer of this new function (new applications, system enhancement) after completing the development of the code.

The components of a package fall into two categories: package objects which are the application files to be installed, and control files which control how, where, and if the package is installed.

Control Files

The control files are divided into two categories: information files, and installation scripts. Some of these control files are required and some are optional, installation scripts are always optional. The installation script must be composed of Bourne shell commands.

Installation script performs customized actions during the installation of your package, especially procedure scripts defining actions that occur at particular points during package installation and removal, and the class action script, which defines a set of actions to be performed on a group of objects. Four procedure scripts are predefined: preinstall, postinstall, prere-move, and postremove.

¹This part does not intend to be a package tutorial, but rather a short introduction to provide the reader enough knowledge about them to understand YASSP internals [13].

Package Objects

These are the components that make up the application. They can be: files (executable or data), directories, named pipes, links or devices.

Package objects are described in the prototype files, one line per single deliverable object. An object description includes its location (pathname, relative or absolute), attributes (owner, group and permission mod), class and file type.

Object classes allow a series of actions to be performed on a group of package objects on installation or removal. The sed's predefined class provides a method for using sed instructions to edit files upon package installation and removal. See Sun's documentation for more details about object classes.

Package Delivery, Installation, Registration and Checking

At the installation of a package, all of its objects are registered in `/var/sadm/install/contents`. Information stored in this file is either static information found in the package's information files (package name, file type of the object, attributes, class), or dynamic information generated at the installation by the `pkgadd` command, like the date, the size and the checksum (`sum(1)`) of the object, which provides a good integrity check but a poor authenticity check.

The accuracy and the integrity of the various objects of an installed package can be verified using the `pkgchk(1)` command. The `-n` option, which tells `pkgchk` not to check volatile or editable files, should be used in a post-installation check. Checks are done on the existence of each object, their recorded attributes, their size, and checksum.

Package information can be updated manually anytime using the `installf(1M)` and `removef(1M)` command.

Packages are the recommended way to deliver Solaris applications. Solaris packages provide tools and methods to install new objects, and modify or delete existing ones through either the action class or the installation script. They also provide a basic mechanism to save files modified and will record all installed files, including their attributes and a basic checksum.

Packages are not perfect but exist on every Solaris system.

SECclean Use of the Package Mechanism

SECclean is the core of YASSP, implementing most of the security tuning. SECclean is built as a package, using the rules we explained in the last section. To implement this security tuning, SECclean will modify the existing Solaris system by adding, deleting, replacing or modifying various files.

Files can be sorted into different groups: created, deleted, replaced, and modified files and startup scripts.

New files, deleted files and replaced files are three categories of files easily understood. Sometimes, it is more prudent to modify an existing file instead of replacing it with our version, as they may be different on each system.

Startup files, or Run Control Scripts as Sun documentation named them, are Bourne shell scripts to control run level changes. Each run level has an associated rc script located in the `/sbin` directory: `rc[0-6S]`.

Run control scripts are located in the `/etc/init.d` directory with links from the corresponding run control scripts in the `/etc/rc[0-6S].d` directories. Due to their specific functions, and requirements, startup files are handled as special cases by SECclean.

Since nothing can be simple, exceptions will exist to this classification. SECclean is compatible with three versions of Solaris, and files needed to be replaced may be dependent of the version. Other files we want to modify will need special care, as these modifications can be more complex. Binary files installed by SECclean need to be handled depending of the architecture of the target. Last, some files may need to be modified only if they exist.

File Management Under SECclean

As explained in the Philosophy section, SECclean must be un-installable, and the state of your system after a SECclean installation and un-installation should be the same as before. All the information associated with a file (its contents, its attributes (owner, group and permission mode) and also the package information related to this file) must be extracted and saved at SECclean installation, so that the removal step will be able to restore it. These backup and restoration procedures have to be created, as packages provide you a guideline on how it should be done, but it is up to the package developer to implement such features. A Bourne shell² library named `cleanlib.sh` provides functions to help deal with package registration, files backup and restoration. To perform any file operations, SECclean uses `cleanlib.sh` functions, as package operations provided by Solaris are very primitive.

Cleanlib provides the following functions:

- `Install_file()`: It installs a file by moving it from its prefixed name to its real name. It keeps a copy of the installed file so that any modification to it can be tracked, and registers the file as part of the SECclean package.
- `Install_RC_file()`: Same function as `Install_file()` but to install a startup script. It also creates the symbolic links needed under the `/etc/rc[0-6S].d` directories.
- `Backup_user_file()`: Copies a file and re-create its path under a backup directory.

²Why Bourne shell? These functions will be used in the package installation scripts, which must be written in Bourne shell, and we should be able to use this library on a core installation.

- `Save_and_move_file()`: Moves the indicated file and its package information into a backup area and then un-registers it from the package database.
- `Disable_RC()`: Removes links to a startup script. It will backup all package information found.
- `Disable_Init()`: Searches for any registered link to a startup script and calls `Disable_RC()` for each of them.
- `RCconfized_Init()`: Modifies a startup script to control its start through `/etc/yassp.conf`. Will also update `/etc/yassp.conf` for each startup script modified.
- `Create_RCconf()`: Initializes the static header of `/etc/yassp.conf`.
- `DE_RCconfized_Init()`: Undoes the modification done by `RCconfized_Init()` on startup scripts.
- `Restore_RC()`: Restores a startup script links from the saved information and updates the package database.
- `Restore_file()`: Restores a file from its backup and updates the package database.
- `Init_preremove()`: Initializes the value for the backup directory for files modified since YASSP installation.
- `Cmp_and_backup_file()`: Checks to see if any previously installed files have been modified and backs them up if so.

To delete an existing file, we first save its current package information, move it to the package save directory and un-register it from its original packages. This is done by the `Save_and_move_file` function.

Replacing a file involves more steps. We must first deliver the new file under a different name, in order to not overwrite the existing one. The convention for SECclean is to install the new file with a name prefixed by 'SECclean_'. Then, we must delete, as explained in the last section, the existing file, move the new file to its real name, un-register the prefixed name from the SECclean package and register the real name in it. Also, as SECclean wants to be able to track any changes to these files after its installation, a copy of the original installed file is saved. These operations are done using the `Save_and_move_file` and the `Install_file` functions.

Files that need to be modified are defined as part of the `sed` class in SECclean prototype. No specific action is required as the associated `sed` script handles the installation and un-installation phase.

For startup scripts, we wanted to offer the systems administrator the choice of running the different startup script present, and we wanted this choice to be as convenient as possible. The solution chosen in SECclean was to modify the existing script so that it will start only if a shell variable is set to YES in YASSP's configuration file `/etc/yassp.conf`. The name

of this shell variable is based on the name of the script, stripped of all non-alphabetic characters and capitalized. The function `RCconfized_Init` is used to modify each original startup script.

The `Disable_Init` function is used to backup and delete any information associated with a startup script. It will, from the name of the startup script under `/etc/init.d`, find all the existing links to it from any `/etc/rc[0-6S].d/` directory, record all these links and the package information associated with them, save the content of the original startup script and delete them all. It is used when we need to replace or delete a startup script.

Cleanlib also provides the reverse functions, used at the removal phase of the package to restore the state and the contents of these files.

Exceptions exist: some startup scripts need further modification, some startup scripts need to be replaced by our version but depending on the operating system version, some files need to be installed only if they were not already present.

All these functions keep track of the list of packages they touch, so that when we are done, we know which changes have to be validated.

Other Tasks Performed by SECclean

In addition to the file operations we described, SECclean also needs to do the following tasks:

User Convenient Backup

We found it very useful and convenient to copy the tree of files SECclean will touch and provide it as an informational copy to the systems administrator. This copy is not used by the package backup mechanism, but provides the systems administrator an easy way to see which and how files were changed. The backup is done by default under `/yassp.bk` (See the 'Putting it all together' section for more details).

Password File Cleanup

The password file will be checked and cleaned up if needed, to lock all non relevant system accounts, change the shell of all locked accounts to a binary which will syslog any use of these accounts before exiting. Password file cleanup is implemented in the `clean_passwd` script, part of SECclean, and can be re-run anytime after SECclean installation.

Operating System Cleanup

When you install Solaris, even if Solaris components are built as packages, you will find some incoherence if you check the package integrity. These incoherencies reflect either files modified during the install, or files registered incorrectly, or both. Using a shell script, which is very OS dependent, SECclean will correct these incoherencies and leave the package database clean.

After, using Casper Dik fix-modes program for Solaris [3], SECclean secures various files permission

and ownership. It does this by removing group and world write permissions of all files/devices/directories listed in `/var/sadm/install/contents`, with the exception of those pre-listed in `fix-modes` program.

Putting It All Together

Installation Process

SECclean makes use of the preinstall and postinstall procedure scripts.

Preinstall

The preinstall script first determines under which directory the SECclean component will be installed. Some components of YASSP, which may be installed later, have pathnames hard coded in their binary³. On the other hand, systems administrators may want to install their software in another place. If `/opt/local` exists and if we can change `dir` to it⁴, SECclean uses `/opt/local` as its default installation path. If we cannot change `dir` to `/opt/local`, but if `/usr/local` exists and we can change `dir` to it, `/opt/local` is created as a symbolic link pointing to `/usr/local`. If none of them exists, `/opt/local` is created as a directory.

Then, the preinstall script initializes the two shell variables `$SECBCK` and `$CLEANUPDIR`, if they are not already defined. `$SECBCK` is used as the pathname under which the user convenient backup will be done. Its default value is `/yassp.bk`. `$CLEANUPDIR` is the path where the scripts and binaries related to the operating system clean-up will be installed. Its default value is `/var/sadm/clean-up`.

Next, the preinstall script defines four shell variables, corresponding to categories of files. The variable `$RCCONF` corresponds to an exhaustive list of startup scripts. Startup scripts are always referred to by their name under the `/etc/init.d` directory. This list is exhaustive, gathering all the startup script names we want to control in all Solaris versions we support. Next variable is `$NRC`, defining the startup script we want to replace with our own OS dependent version. Three startup scripts are part of this list: `inetsvc`, `inetinit` and `networks` (only on Solaris 8). They are the scripts driving the network setup at the boot time. The last two variables defined are the list of files to be deleted (`$SD`) and the list of files to be replaced (`$SA`). All these variables are recorded in a file (`$PKGSAVE/.PROC_Init Var`), so that the other scripts used in SECclean installation or removal can share the same lists. For an up-to-date list of the files defined in each of these variables, please consult <http://www.yassp.org>.

The user convenient backup is the last thing done by the preinstall: all files SECclean intended to touch will be copied under `$SECBCK/Before_‘date +%Y.%m.%d-%H.%M.%S’`.

³An example is the RCS package, compiled with the GNUdiff: RCS binary has the pathname of GNUdiff hard coded. The default pathname we used is `/opt/local`

⁴`/opt/local` can be a symbolic link to any existing directory.

Installation

It is at the installation phase that all the package objects defined in SECclean will be installed. Objects which correspond to files we want to replace have been registered and will be installed with a name prefixed by `'SECclean_'`. For objects dependent on the operating system version (`inetsvc` for example) or on the architecture of the host (binary), all possible instances were registered and will be installed using the OS version or the architecture as a suffix.

All sed scripts associated with any package objects part of the sed class will be then executed, and the targeted file modified.

Postinstall

The postinstall script is the most complex one.

After reading the `PROC_Init_Var` file created by the preinstall to learn the value of the different categories of the file, it first disables startup files we will replace later, by calling the `Disable_Init` function.

Next, all the files listed in `RCCONF` are modified, using the `RCconfized_Init` function. For each of them, a sed script is generated and applied to them. From the result of which startup files were present on this system, and from some static information, the `yassp.conf` file is generated.

All the files we will delete or replace are backed up and moved out using `Save_and_move_file`. At this step, SECclean is done modifying existing packages, and it can now close and validate all the removals done to the package database.

The next steps deal with SECclean package objects. For files that are dependent on the operating system version or on the machine architecture, the right version is kept and others are deleted and unregistered. SECclean package information is updated and the database is closed. All the files we want to replace, that have been installed using `'SECclean_'` as a prefix in their name, are renamed to their right name. The new `inetinit`, `inetsvc` and `networks`⁵ startup scripts are installed and symbolic links are created from the appropriate `/etc/rc[012].d` directories, and special cases are handled for specific files. SECclean database is then closed again, to validate all these changes.

We are now done with file installation and package database update. Tuning to some newly installed files like the `/etc/system` is done depending on the operating system version, `syslog` files used by the newly installed `syslog.conf` file are touched, and the password file is cleaned up by calling `clean_passwd`.

The last thing done is the operating system clean up, which includes the package database correction and running `fix-modes`.

We are done: SECclean is installed.

⁵These startup scripts are treated as a special case: we want them to execute the strict minimum instead of handling all the cases planned by Sun (DHCP for example). The new scripts provided are always based on the original ones. Setting variables in `yassp.conf`, can restore the original Sun behavior of these scripts.

Removal

The removal phase has two main tasks: restoring the state your workstation as it was before SECclean installation, but also keeping a copy of any files, installed by SECclean, and modified by the systems administrator so that these modifications will not be lost. It will use the preremove and postremove procedure scripts to implement these tasks.

Preremove

After reading the PROC_Init_Var file created by the preinstall to learn the value of the different categories of files, it checks all files replaced by SECclean against the originals, and keeps a backup copy under /var/tmp/SECclean.Backup_\$\$ if they have been modified. Next, it makes a copy of the cleanlib.sh library, as this file will be removed in the removal phase, and the postinstall needs it.

Removal

All files registered as part of SECclean prototype are deleted. Sed scripts associated to package object of the sed class will be executed to undo any modifications done.

Postremove

After reading the PROC_Init_Var file created by the preinstall to learn the value of the different categories of files, it will unregister from the package database all files registered as part of SECclean in the postinstall script, listed in the \$SA and \$NRC variables, and close the database.

Next step is to restore the original contents of files SECclean has deleted or replaced, and re-register them in the package database with their original attributes.

The modifications to control the start of the startup scripts are undone, and the package database is closed to validate these changes.

Last, the copy of the cleanlib.sh library is deleted.

How We Tested It

This complex process was validated by installing (and un-installing) YASSP on various hosts, for testing as well as being part of the day-to-day work.

The final state after the installation is tested using tools like titan [8], ISS ... and of course we asked the beta testers to use their expertise.

The SANS team is also very helpful for advice or consensus on proposed changes, or for pointing out existing documentation and tools [2, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14].

Particular attention was given to test the package removal function. Testing can be time consuming, especially when the package is broken and corrupts the package database. Use of VMware virtual machine running Solaris for testing is a big time saver.

YASSP Other Goodies

YASSP also offers to install a set of useful and common tools, built as separate packages. It includes:

- GNUrcs: RCS 5.7 and diff 2.7 [GNU] to ease version control of your files.
- GNUgzip: gzip 1.2.4a [GNU]: file compression.
- PARCdaily: Some daily scripts, logs rotation, backup and RCS for system files... Need GNUgzip and GNUrcs. It is mainly an example to show people what they can do.
- WVTcpd: tcp_wrappers 7.6 + rpcbind 2.1 [Wietse Venema]: tcp wrapper, and Wietse's version of rpcbind compiled with tcp wrapper library, for people who need to run RPC.
- PRFtripw: Tripwire 1.2 [Purdue Research Foundation of Purdue University]: Integrity checking tools
- SSHsdi: version 1.2.30-SDI, build with SecurID support (See <ftp://ftp.parc.xerox.com/pub/jean/sshsdi/README>).

All these packages will, when installed, take all the necessary steps to be ready for use. For packages relying on configuration files, some proposed configuration files are provided, and they will be installed as the default configuration file if they were not already present.

YASSP Installation

What Can Go Wrong During the Install

From our experience, nothing should go wrong if it is a fresh Solaris install. For an installation done on an existing system, you may find some conflicts during the package installation with files or directories.

Example of Installation

```
# uncompress yassp.tar.Z
# tar xvf yassp.tar
# cd yassp
# ./install.sh
... check and modify all
    the configuration files ...
# reboot
```

A commented installation log can be found in Appendix 2.

What May Be Broken After?

SECclean and YASSP defaults are to disable (or to 'secure') most of the existing services. If you are planning to install YASSP or SECclean on an existing server, you must know what application(s) are running on it, and what network services and system resources are being used. Your work after the YASSP install will be to re-enable these services, and only these services, to have a system where you know why any process or service is running. YASSP will help you by providing some tools to monitor these changes and re-enable some security features after its installation, but YASSP cannot understand your setup as you can do.

Also, if your system had non-standard applications or configurations, SECclean may have missed some tuning specific to your application.

What To Do After?

Most of YASSP configuration is handled by a single text file `/etc/yassp.conf`. A man page, `yassp.conf(4)` <http://yassp.man/yassp.conf-4.html>, describes the different tuning that can be done through this file. A more generic description of what to do after the installation, not focused on SECclean, is maintained at: <http://yassp/after.html>.

And do not forget to send us feedback about YASSP; we are eager to receive comments, suggestions or just acknowledgement that people are using it.

Where Are We Now?

As of October 10, 2000, beta#12 of YASSP is released on <http://www.yassp.org>. It has been tested and supports Solaris 2.6, 2.7 and 8, under sparc architecture. Beta#12 is used widely⁶, and is the release candidate for v1.0 [17].

YASSP is built on top of two independent components:

- The core package, named SECclean, implementing the OS security modification, which also includes correcting the incoherence left after a standard Solaris install and modifies some filemodes to make the file hierarchy more secure.
- A set of additional packages, provided as an optional install as they represent some commonly used features not implemented by Solaris.

Systems administrators can use these tools choosing either the YASSP `install.sh` script included on the YASSP tar file, or to run each package or script manually.

The default `install.sh` script of YASSP will cleanup the Solaris install, modify the filemodes to secure various files and directories, install the core package and propose to install a set of additional packages.

After installing the core package, SECclean, the result is a system quite closed and secure, where most of the services have been turned off, minimal processes are running, and with some security tuning applied at various levels of the operating system, from the network stack to the XDM configuration file.

Future

Still a lot to do, from gathering more feedback, to porting it to new Solaris releases, and improving the web documentation.

⁶Widely is vague, but it is hard to be more precise. Since the YASSP site was announced, the logs show the number of downloads, but unfortunately, we have received very little feedback.

Two main directions can be guessed from the talk in YASSP mailing list:

- Develop new tools to help monitor the system after the initial installation, to check that no modifications happen and be able to re-apply some security settings on-demand.
- Develop a tool to manage some more restrictive file permissions, especially for SUID/SGID binaries, as an option.

Conclusion

The YASSP project has taught us a lot on Solaris internals, both from a security point of view and about the Solaris package mechanism. The complexity added by the way YASSP deals with files and package information has shown its strength as it provides a robust un-installation mechanism and eases a lot YASSP maintenance for any new version of Solaris.

YASSP is used by various organizations, in various contexts from end user workstations to exposed servers. Feedback we received shows us that it is a useful tool that people appreciate. YASSP is young, and will continue to be improved, but we have to be careful to keep it focused on securing a Solaris host, and keep its size reasonable so that we can maintain it

Last, we would like to thank all the people who have participated in the YASSP mailing list or have sent us feedback. Please continue.

References

- [1] Bastille Linux, <http://bastille-linux.sourceforge.net/>.
- [2] Boran, Seán, *Installing a Firewall bastion host: Hardening Solaris* http://www.boran.com/security/sp/solaris_hardening3.html.
- [3] Dik, Casper, *Great Solaris tools (fix-modes) and the Solaris 2 FAQ*, <http://ftp.wins.uva.nl/pub/solaris>, <http://www.science.uva.nl/pub/solaris/solaris2.html>.
- [4] Kim, Gene and Gene Spafford, *Tripwire: Free version V1.2*, <ftp://ftp.cerias.purdue.edu/pub/tools/unix/ids/tripwire>.
- [5] Noordergraaf, Alex and Keith Watson, *Network Settings for Security*, <http://www.sun.com/software/solutions/blueprints/1299/network.pdf>.
- [6] Noordergraaf, Alex and Keith Watson, *Minimization for Security: A Simple, Reproducible and Secure Application Installation Methodology*, <http://www.sun.com/software/solutions/blueprints/1299/minimization.pdf>.
- [7] Pomeranz, Hal, *Building Bastion Hosts With Solaris: Step by Step*, <http://www.deer-run.com/solaris.html>.
- [8] Powell, Brad, Matt Archibald, and Dan Farmer, *The Titan Project*, <http://www.fish.com/titan/>.
- [9] *Sabernet Solaris Security Guide*, <http://www.sabernet.net/papers/Solaris.html>.

- [10] *Solaris Security FAQ*, <http://www.sunworld.com/common/security-faq.html>.
- [11] Spitzner, Lance, *White papers*, <http://www.enteract.com/~lspitz/papers.html>.
- [12] Sun, *Solaris 8 System Administration Guide, Volume 1*, <http://docs.sun.com/ab2/coll.47.11/SYSADV1/@Ab2TocView?Ab2Lang=C&Ab2Enc=iso-8859-1>.
- [13] Sun, *Application Packaging Developer's Guide*, <http://docs.sun.com/ab2/coll.45.13/PACKINSTALL/@Ab2TocView?Ab2Lang=C&Ab2Enc=iso-8859-1>.
- [14] University of Waterloo, *Security: How to Documents*, <http://ist.uwaterloo.ca/security/howto/>.
- [15] Venema, Wietse, *Tools and papers*, <ftp://ftp.porcupine.org/pub/security/index.html>.
- [16] Vöckler, Jens-S., *Solaris 2x – Tuning Your TCP/IP Stack and More*, <http://www.rvs.uni-hannover.de/people/voeckler/tune/EN/tune.html>.
- [17] *YASSP Mailing list archive*, <http://www.theorygroup.com/Archive/YASSP/>.

History of the YASSP project

A Long Time Ago...

Back in 1997, we faced the task of having to deploy firewall and external servers in various locations around the world. Even if the configuration was standard and well known, we realized quickly that, as the local Unix skills were unknown, we needed to help systems administrators and provide a way to install these servers so that we could be pretty comfortable that the result of this installation was what we expected to see.

We started to write down some guidelines on how to install and then secure these servers. That was perfect for experienced systems administrators, but we quickly realized that something written does not mean that people will read it, and that this guideline was written for people understanding the context of the installation (skilled in network, Unix, Solaris), which was not always the case at these remote sites.

We translated these guidelines to shell script, very primitive, which was supposed to do all the work automatically. It required good Unix skills from the systems administrator running it, was very touchy with the existing state of the server, which had to be a fresh installed OS, and was destructive in the sense that files were replaced without backup. There was no way to undo the damage caused by it, neither was there a way to use it to perform a different task than installing an external server as defined per our organization.

Xerox PARC

In 1999, the author came to Xerox Palo Alto Research Center and was able to allocate more time to this project. The idea was to enhance the existing baseline and enable its use on end-user workstations.

The shell script was ported to the Sun package format, allowing it to be un-installed and integrated into our (PARC) default installation process.

This package grew, including lots of suggestions from its users (mainly Xerox people) and new features or tuning gathered from various sources.

At this time, it provided a way to harden a fresh Solaris installation to be used mainly on an exposed server (most of the services will be turned off), was un-installable if necessary. A public version was available on our external ftp site and was used by various people. Basic documentation existed on the same ftp server.

SANS

With the SANS Institute, we created a team to work on tools for securing Solaris. This team (10 people at the beginning, about 160 now) was open to anyone who committed to spend time testing or reviewing existing tools and providing feedback.

It drove some major changes on the existing tools.

First, the scope was extended: it should still focus on exposed servers, but should be also suitable for end-user workstations.

Also, it was modified to be permissive on the expected state of the operating system and able to be installed on existing install rather than requiring a fresh Solaris install.

Some recommended security tools were added to it, implementing additional features not available in Solaris.

At last, and on top of all the bug corrections and enhancements suggested by this team, the tool was modified to be more verbose, have better documentation and provide the systems administrator an easier way to configure the desired security level.

Of course, some problems were raised and some nice verbal argumentation happened, but life would be too quiet without that.

Why Does It Still Exist?

At the time it was started, there were not many tools available to secure Solaris, but now, various other tools exist (Titan, Bastille-Solaris), and people might wonder why we spend our energy on creating and maintaining a separate tool rather than merging our efforts with existing tools.

The first reason to keep YASSP alive was that it existed, was working, was used by people and was useful to them if we believe the feedback we received at the time.

Also YASSP deals correctly with Solaris packages, and will keep the database clean. This part was unique to YASSP, as well as the unique configuration file it offers.

Last, we found that for some systems administrators, it was less confusing to have the minimum

interaction with the script, especially with less Unix experienced systems administrators. These systems administrators, with little or no Unix/Solaris experience, may be required to install Solaris server in a hostile environment. YASSP will do that by default.

Security is a field where having a choice of tools is a plus. More choices provide flexibility and more appropriate solutions.

Appendix 1: System with YASSP installed

```

Rebooting with command: boot
Boot device: disk File and args:
SunOS Release 5.8 Version Generic 64-bit
Copyright 1983-2000 Sun Microsystems, Inc. All rights reserved.
configuring IPv4 interfaces: hme0.
Hostname: zeta
Tweaking Solaris TCP/IP: Solaris 7 or above (excellent)
    tweaking separate connection queues
    tweaking against SYN flood symptoms
    tweaking timeouts
    tweaking pMTU discovery interval and common timers
    tweaking misc. parameters
    applying security tweaks...
    tweaking windows, buffers and watermarks
done.
The system is coming up. Please wait.
checking ufs filesystems
/dev/rdisk/c0t0d0s5: is clean.
Setting netmask of hme0 to 255.255.255.0
syslog service starting.
sshd starting.
The system is ready.
WARNING: To protect the system from unauthorized use and to ensure that
the system is functioning properly, activities on this system are
monitored and recorded and subject to audit. Use of this system is
expressed consent to such monitoring and recording. Any unauthorized
access or use of this Automated Information System is prohibited and
could be subject to criminal and civil penalties.
zeta console login: root
Password:
Last login: Wed Jul 19 19:07:06 on console
This computer system for authorized use only
# ps -eaf
    UID    PID  PPID  C    STIME TTY      TIME CMD
    root      0      0  0  22:49:18 ?        0:15 sched
    root      1      0  0  22:49:19 ?        0:00 /etc/init -
    root      2      0  0  22:49:19 ?        0:00 pageout
    root      3      0  0  22:49:19 ?        0:00 fsflush
    root    212      1  0  22:49:45 console 0:00 -sh
    root    173      1  0  22:49:43 ?        0:00 /usr/sbin/syslogd
    root    185      1  0  22:49:44 ?        0:04 /opt/local/sbin/sshd
    root    227    212  0  22:53:42 console 0:00 ps -eaf
    root    168      1  0  22:49:42 ?        0:00 /usr/sbin/cron
# netstat -an
UDP: IPv4
    Local Address      Remote Address      State
    -----
    *.514
    *.
    Idle
    Unbound
TCP: IPv4
    Local Address      Remote Address      Swind Send-Q Rwind Recv-Q  State
    -----
    *.
    *.
    0      0 24576      0 IDLE
    *.22
    *.
    0      0 32768      0 LISTEN
    *.
    *.
    0      0 32768      0 IDLE
TCP: IPv6
    Local Address      Remote Address      Swind Send-Q Rwind Recv-Q  State      If
    -----
    *.
    *.
    0      0 24576      0 IDLE
# pkgchk -n

```

Appendix 2: Installation example

Commands typed by the systems administrators are in fixed-width bold; comments are in *italic roman* typeface.

```
zeta console login: root
# ./install.sh
```

Running YASSP installation script.

```
<...>
```

Type the package list you want to install or hit return to accept the default:

```
SECclean GNUrcs GNUgzip PARCdaily WVtcpd PRFtripw
```

```
<return>
```

We chose to install all packages proposed.

```
Do you want to install SSH (See the SSH-COPYING file for the SSH
license)?: [y|n] (n) y
```

We want SSH.

```
YASSP will install: SECclean GNUrcs GNUgzip PARCdaily WVtcpd PRFtripw SSHsdi
Installing the various package:
```

```
=====
SECclean
```

SECclean installation start.

The pre-install runs, initialize some variable and back-up files it will modify.

Using /opt/local as the root dir.

Backing up all files under /yassp.bk/Before_2000.07.19-22.35.53:

```
<... Long list of files ...>
```

Pre-install is done.

The install runs: files declared in the prototype are installed silently. Files, part of the sed class in the prototype, are modify by the associated sed script.

```
Modifying <...>
```

```
***
```

The postinstall start. It first reads the variables stored by the pre-install.

The postinstall script is silently running. It may take a while on slow machine. Just be patient

Disabling init files we will replace later.

```
Disabling startup files: inetdsvd inetdinit network
```

Modifying startup files to be controlled by yassp.conf.

```
Modifying Startup files to use /etc/yassp.conf: <... list of all startup
files modified ...>
```

Creating /etc/yassp.conf, as we know now which startup file were modified.

```
Creating your default /etc/yassp.conf
```

Saving (in the package's save directory) and deleting files.

Some of them will be replaced by SECclean's own version later.

```
Saving files: <... list of files backed up ...>
```

We have unregistered (removef) all the files we deleted from the package database.

We must now close (removef -f) these open packages.

```
Closing the package we touched <... package name list ...>
```

These are the files that will be replaced by SECclean version. They have been installed (as part of SECclean's prototype file) as /path/name/SECclean_{name_of_the_file}, and are registered under this name as part of SECclean package. We must first unregistered (removef on SECclean) them.

```
Updating SECclean package DB: <... list of file ...>
and close SECclean (removef -f SECclean)
```

```
Closing SECclean DB
```

Move the files from their SECclean_{name} to {name} and register them as part of SECclean (installf)

```

Replacing: <... list of files ...>
    OS specific startup files: chose the right version.
Choosing the right startup files: /etc/init.d/inetsvc
/etc/init.d/inetinit /etc/init.d/network for your OS: Solaris 5.8
    Replacing them, registering (installf) as part of SECclean
    package, and creating the sym-link.
Replacing Special startup files: /etc/init.d/inetsvc
/etc/init.d/inetinit /etc/init.d/network and creating the symlink
    Closing (installf -f SECclean) SECclean.
Closing again SECclean DB
    Specific OS tuning: for Solaris 8, no priority_paging
Tuning /etc/system to comment out priority_paging
    Running clean_passwd
Cleaning the passwd file...
Disabling UID 0 account(s):
Disabling system account(s):
    daemon bin sys adm lp uucp nuucp listen nobody noaccess nobody4
Deleting account(s):
root identity will be changed to "Root at zeta"
password and shadow files saved under /etc/passwd.Old and /etc/shadow.Old
    Doing the OS cleanup: fix-modes is run first...
Doing the OS Clean-up
Running fix-modes 2.6 2000/01/13 14:13:35 casper
fix-modes done, log file under: /var/sadm/clean-up/clean_up.log
    Then we correct well-known incoherence in the Sun installation
clean-up the contents database
cleanup done, log file under: /var/sadm/clean-up/clean_up.log
Running /usr/lib/makewhatis /opt/local/man
    We are done with SECclean, echo the summary.

=====
SECclean installation has finished.
Changes to the file-system and package database are documented in:
    /var/sadm/clean-up/clean_up.log
All changed or replaced files are archived in
    /yassp.bk
If existing, the crontab for the users:
    lp adm uucp root
has been deleted. Please, re-enable manually the entries needed
Backup for the crontab files are under:
    /yassp.bk/var/spool/cron/crontabs/
To finish hardening, this host must be rebooted.
However, you should first check that /etc/yassp.conf is configured
to your requirements. See also yassp(1) and yassp.conf(4).
=====

Installation of <SECclean> was successful.
=====
GNUrcs
=====
Installation of <GNUrcs> was successful.
=====
GNUgzip
=====
Installation of <GNUgzip> was successful.
=====
PARCdaily
=====
Modifying /usr/lib/newsyslog
Installation of <PARCdaily> was successful.
=====

```

WVtcpd

```

=====
      tcp_wrappers add some example of how to use it in comment in /etc/inetd.conf
Modifying /etc/inet/inetd.conf
      and install some default configuration file if they were not present.
Creating /etc/hosts.deny from the distribution file
*** Please configure it!
Creating /etc/hosts.allow from the distribution file
*** Please configure it!
Installation of <WVtcpd> was successful.
=====

```

PRFtripw

```

=====
      Default configuration file is created if it was not present.
Creating /secure/tripwire/tw.config from the distribution one:
/secure/tripwire/tw.config.Dist
*** Please configure it!
you may use tripwire now.
Type: "cd /secure/tripwire/; ./tripwire -i 2 -initialise -c tw.config"
      to create a new database,
Use "cd /secure/tripwire/; ./tripwire -q -i 2 -c tw.config"
      to check,
***** SAVE YOUR DATABASE IN A SECURE PLACE *****
Installation of <PRFtripw> was successful.
=====

```

SSHsdi

```

=====
Modifying /etc/syslog.conf
      default configurations files are created if they were not present.
Creating /etc/ssh_config from the distribution file
*** Please configure it!
Creating /etc/sshd_config from the distribution file
*** Please configure it!
      We generate the keys if they were not present.
Initializing random number generator...
<...>
Your public key has been saved in /etc/ssh_host_key.pub
ssh has been installed.
run '/etc/init.d/sshd stop;/etc/init.d/sshd start' and restart syslogd
to use the new binaries/configuration
Installation of <SSHsdi> was successful.
      YASSP install is done, now recreate the whatis database if it was present.
Rebuilding the whatis database
YASSP is installed.
Most of these changes will take action at the next reboot.
**** YOUR WORK IS NOT DONE YET ****
*) Edit and configure /etc/yassp.conf
*) Edit and configure /etc/hosts.deny /etc/hosts.allow
*) Edit and configure /etc/sshd_config /etc/ssh_config
*) Read http://yassp.parc.xerox.com/after.html
   and the papers linked under http://yassp.parc.xerox.com/ref.html
*) make any additional changes/software installation
*) CREATE YOUR tripwire DATABASE AND SAVE IT!!!

```

```

Type:
vi /etc/yassp.conf /etc/hosts.deny /etc/hosts.allow /etc/sshd_config
   /etc/ssh_config ; cd /secure/tripwire; ./tripwire -i 2 -initialise -c
   tw.config; cp /secure/tripwire/databases/tw.db_zeta TO_A_SECURE_PLACE
      ***YOUR feedback*** is important: please send comments or flame to:

```

sansro@sans.org, chouanard@parc.xerox.com
with "YASSP" in the subject

reboot

SubDomain: Parsimonious Server Security

Crispin Cowan, Steve Beattie, Greg Kroah-Hartman, Calton Pu, Perry Wagle and Virgil Gligor¹
– WireX Communications, Inc.

ABSTRACT

Internet security incidents have shown that while network cryptography tools like SSL are valuable to Internet service, the hard problem is to protect the server itself from attack. The host security problem is important because attackers know to attack the weakest link, which is vulnerable servers. The problem is hard because securing a server requires securing every piece of software on the server that the attacker can access, which can be a very large set of software for a sophisticated server. Sophisticated security architectures that protect against this class of problem exist, but because they are either complex, expensive, or incompatible with existing application software, most Internet server operators have not chosen to use them.

This paper presents SubDomain: an OS extension designed to provide sufficient security to prevent vulnerability rot in Internet server platforms, and yet simple enough to minimize the performance, administrative, and implementation costs. SubDomain does this by providing a *least privilege* mechanism for *programs* rather than for users. By orienting itself to programs rather than users, SubDomain simplifies the security administrator's task of securing the server.

This paper describes the problem space of securing Internet servers, and presents the SubDomain solution to this problem. We describe the design, implementation, and operation of SubDomain, and provide working examples and performance metrics for services such as HTTP, SMTP, POP, and DNS protected with SubDomain.

Introduction

Common server operating systems such as Linux, Windows, Solaris, etc. are subject to vulnerability rot as security vulnerabilities (i.e., implementation bugs) are discovered in the component software of these operating systems. For instance, a buffer overflow discovered in the BIND domain name server [15] allowed remote attackers to gain root privileges on a variety of system platforms, and a similar vulnerability in Microsoft's IIS (web server) [21] allows remote attackers to gain administrative control of Windows servers. The recommended defense for general purpose servers is to keep the host system up to date with vendor patches to close these vulnerabilities.

However, many of these systems are being pressed into use as the basis for *server appliances*: servers intended for largely unattended operation by unskilled users. But because these operating systems are subject to vulnerability rot, they need to be frequently upgraded with vendor patches. While this is an acceptable approach for general purpose servers (where a skilled system administrator is expected to maintain the system) it is not acceptable to appliance users, who expect a device with the maintenance factor of a toaster.

The classical security solution to vulnerability rot is the notion of *least privilege*: the technique of granting subjects in a system precisely the capabilities they need to perform their function, and no more [33]. Effective use of least privilege *minimizes* the potential damage that results when a trusted program is penetrated by minimizing the degree to which the program is trusted.

Security architectures that provide least privilege mechanisms exist, but because they are either complex, expensive, or incompatible with existing application software, appliance vendors have not chosen to use them. Existing defenses entail these complexities precisely because they were designed to handle the generality of a general purpose server, and thus must deal with user least privilege. This generality complicates the least privilege abstraction, making the enforcement mechanism more complex to implement and use.

This paper presents SubDomain: an OS extension designed to provide sufficient security to prevent vulnerability rot in server appliances, and yet simplify as much as possible to minimize the performance, administrative, and implementation costs. SubDomain does this by providing a least privilege mechanism for *programs* rather than for *users*. The security restrictions *complement* the system's existing permissions, allowing a program to be secured independent of who may be using the program. This notion is especially effective on server appliances, and enables program-specific confinement information to be distributed

¹Cowan, Beattie, and Pu are formerly of the Oregon Graduate Institute, where much of this work was done. Cowan and Beattie are now with WireX Communications, and Pu is now with the Georgia Institute of Technology. Wagle is still with the Oregon Graduate Institute, and Gligor is with the University of Maryland. Kroah-Hartman is with WireX.

with the program (see the section on SubDomain compatibility).

By specifically addressing least privilege for programs, we can provide a mechanism that has a relatively small implementation and simple operation. Small implementations are important for security systems to avoid vulnerabilities due to bugs in the enforcement mechanism itself. Simple operation is important for security systems to avoid misconfiguration. Even more so than in most OS design issues, parsimony is critical to security [33] making SubDomain's relative simplicity of design and implementation an important feature.

We present the SubDomain notation for recursively specifying the sub-domain of resources available to a software component, our implementation of SubDomain as an enhancement to the Linux kernel, our application of SubDomain confinement to several example applications, performance metrics on the cost of SubDomain confinement, and our analysis of the security of a system protected by SubDomain.

The challenge of supporting least privilege is to provide a specification system that is expressive enough to specify privileges that are actually minimal, is convenient enough that administrators can reasonably specify least privileges, and yet preserves compatibility and performance. While SubDomain strives for simplicity relative to other least privilege mechanisms, it provides for finer granularity least privilege in one important regard: SubDomain can confine *arbitrary* software components, at a finer granularity than the native OS process, i.e., procedures and modules. This is especially important for component-based services such as Apache [9] and its loadable modules (see the section on confinement).

The rest of this paper is organized as follows. The next elaborates on the problem of vulnerable/buggy software, and describes the abstract solution of *least privilege* to minimize the potential damage due to attacks against vulnerable software. Readers familiar with least privilege can skip ahead to the third section, which describes the SubDomain security enhancement, and how it advances over previous least privilege mechanisms by providing finer granularity, and simplifying the problem of confining suspect *programs*. The fourth demonstrates SubDomain's compatibility by confining assorted software components, including *sub-process modules*. The fifth section presents the performance costs of SubDomain confinement. The subsequent section 6 describes related work specifically addressing the problem of confining suspect programs. The final section presents our conclusions.

The Problem: Vulnerable Programs and Least Privilege

Many security vulnerabilities result from bugs in "trusted" programs. A "trusted program" is a

program that runs with privilege that some attacker would like to have, and the program fails to keep that trust if there is a bug in the program that allows the attacker to acquire that privilege. Some examples include:

Buffer Overflows: Many privileged programs contain "buffer overflow" vulnerabilities, a problem endemic to C programs that provide poor bounds checking on user-supplied input. Buffer overflows are very common [18, 19] and very dangerous [32, 29], allowing attackers to take control of programs from an anonymous node on the internet.

Race Conditions: Many privileged programs also contain "race condition" vulnerabilities. Here, the problem is that careless root privileged processes create files without adequate checking for the prior existence of the file. The problem is that the attacker can create a symbolic or hard link in the file system between the time the privileged program checks for existence and the time it creates the file, with the result that the root program unwittingly uses its authority to corrupt some other critical file [12].

Special Character Processing: While few root privileged programs are written in shell scripting languages, many other programs with "interesting" privileges are written as shell scripts, especially CGI/ PERL [41] programs for processing web forms. CGI programs run with the authority of the web server, and must process arbitrary input from arbitrary users. If the attacker can provide input (using creative URLs) to a CGI program that yields control to the attacker, then the attacker can gain control of the web server, e.g., the PHF program (included in early NCSA and Apache web servers) allowed the attacker to present a URL to the web server that would cause PHF to start an xterm on the attacker's display [14].

Note that while "trusted" usually refers to highly privileged processes (e.g., root processes) they can actually be processes with *any* privileges that the attacker wants but does not have. The general case is that any program installed on a computer that processes input from potentially hostile users becomes a potential vulnerability. Eliminating these vulnerabilities requires some form of assurance that the program in question does not contain exploitable bugs, but this kind of assurance is problematic. Some classes of bugs, e.g., buffer overflow vulnerabilities, can be eliminated through various compiler techniques [39, 17, 26, 37]. Other forms of vulnerabilities are undetectable at compile time, e.g., race conditions [12] and general logic errors.

The only way to assure the complete absence of a security vulnerability in a program is through expensive manual verification. In the absence of such verification, one must either suffer the risk of potential vulnerabilities, or *contain* the potential damage. Note that

the activities we seek to constrain are “those that cause damage to the system,” i.e., *safety properties* [1] with respect to *integrity*. We are not addressing other security issues, such as *information flows* [27] that might disclose secrets. Readers already familiar with least privilege mechanisms can skip to the next section for a description of SubDomain, our contribution to the field.

The Solution: Least Privilege

The classic solution to the problem of unknown security vulnerabilities is to perform each activity with the *least privilege* required to complete that task [33]. While this does not stop exploitation of these vulnerabilities, it does contain the damage as much as possible. An attacker who gets control of a least privilege process can, at most, read secrets and corrupt data that the exploited process has access to, and no more.

The challenge of supporting least privilege is to provide a sufficiently *fine-grained* mechanism to specify privileges that are actually minimal, while also preserving compatibility and performance. It is conceptually simple to divide system privileges into fine-grained units and then attribute the exact required privileges to a given activity, but the result of such an approach is specification notation that is tedious to maintain (breaking compatibility) and an enforcement mechanism that is slow (breaking performance).

Practical least privilege therefore involves abstracting the system resources to expedite least privilege specifications. Matching least privilege abstractions to native OS resources in turn enables efficient least privilege enforcement. Least privilege is also a useful notion in managing *user* privileges, leading many systems to combine least privilege for users and programs into a single mechanism, as described in the next subsection.

However, if the problem is bugs in programs that can be accessed by completely untrusted users, then user-oriented least privilege mechanisms may become awkward or inadequately expressive. The section on users and rolls describes some more elegant approaches to using user privilege mechanisms to confine suspect programs. The third section discusses SubDomain, our OS security enhancement that particularly address the problem of least privilege for programs, and a penultimate section discusses related work specifically aimed at program confinement.

Using User Privileges to Confine Programs

Least privilege for users is a classic way of structuring a system, and many operating systems provide facilities for constraining the privileges of a given user. User-oriented least privilege facilities can be adapted to confining collection of programs by creating a *synthetic* user, and then running the program as that user.

The classic example is the UNIX *setuid* facility: the *setuid* bit for an executable file indicates that the program runs with the privilege of the owner of the

file instead of the privilege of the invoking user. Often this is used to create *setuid* root programs that provide controlled access to protected resources by expanding the privileges the program runs with to be all of root's privileges. To use *setuid* to confine a program to a smaller set of resources, a new synthetic user can be created that has those privileges, e.g., *nobody*. Programs can then be made *setuid* *nobody* to confine their actions to a small set of privileges.

One limitation to this approach is that all user-IDs, even synthetic user-IDs, can access all files on the system that permit “other” accesses. Another limitation to this approach is that only root can create new user-IDs. The result is that normal users cannot construct ad hoc “sandboxes” for programs that they may choose to install and run. Users are then left with their choice of:

- beg the system administrator to create a new user-ID for them,
- do not install software that is not trusted,
- run untrusted software without protection, none of which is very appealing.

So in principle, synthetic user-IDs and the *setuid* mechanism can support least privilege for programs, but in practice it forces root to do all the work. Therefore this technique is rarely deployed, people run untrustworthy software with much more privilege than is necessary, and suffer the consequent security risks.

Users and Roles

Because synthesizing user-IDs is awkward, the notion of a *role* emerged. A role is a collection of related privileges [2]. In 1986, Bobert and Kain introduced the notion of *type enforcement*: objects (files) are assigned to *types*, subjects (processes) are assigned to domains, and tables determine which domains have access to which types. Badger et al expanded on this notion [7, 8]. In a similar vein, *role-based access control* (RBAC) [22, 34] assigns users to roles, and then grants privileges to the roles.

Similar to the *setuid* approach described previously, roles can be pressed into service confining programs to a least privilege set of resources by assuming a specific role just prior to executing the program. While using roles to confine programs is more elegant than synthesizing user-IDs, it is still fundamentally overloading a user-oriented access control mechanism to manage software defects. In the next section, we describe our mechanism to specifically address the problem of vulnerable software.

SubDomain Security: Recursive Component Confinement

SubDomain is a kernel extension designed specifically to provide least privilege confinement to suspect programs. SubDomain allows the administrator to specify the *domain* of activities the program can perform by listing the files the program may access, and the operations the program may perform.

SubDomain restrictions *complement* the native access controls, in that SubDomain never *expands* the set of files a program may access, i.e., any file access must pass the native access controls and the SubDomain restrictions before access is granted. Thus SubDomain confinement makes a program monotonically safer to run.

The next subsection describes the SubDomain notation and semantics. Then, the subsequent subsection explains how SubDomain leverages work in safe programming models like *proof-carrying code* [30] to achieve component confinement below the granularity of a native process. The final subsection describes the SubDomain implementation.

SubDomain Notation & Semantics

Figure 1 shows a trivial SubDomain specification, in which the foo program is given read access to the /etc/readme file, write access to the /etc/writeme file, and execute access to the /usr/bin/bar file. When ever the program foo is run, by any user, it is restricted to access these specified files with these modes. SubDomain profiles can also grant access to directories through simple globbing, i.e., the profile in Figure 1 grants the foo program to all files in /mydir.

```
foo {
  /etc/readme      r ,
  /etc/writeme     w ,
  /usr/bin/bar     x ,
  /mydir/*         r ,
}
```

Figure 1: Trivial subDomain.

```
foo {
  /etc/readme      r ,
  /etc/writeme     w ,
  /usr/bin/bar     x +{/etc/otherwrite w} ,
  /usr/bin/baz     x -{/etc/writeme w} ,
}
```

Figure 2: Relative SubDomain

The x (execute) capability is of particular importance: what restrictions should apply to the child process? By default, the child process inherits the parent's SubDomain, preventing the confined program from "escaping" its confinement by executing an unrestricted child process. However, sub-components of an application may require different capability sets than the application as a whole. For instance, games only need strong privileges to initialize video controllers, and mail delivery agents only need strong privileges to actually write to a user's mail box. Thus child programs can be given different constraints by specifying a *relative subdomain*, denoted by a x followed by a + or - followed by a SubDomain specification. For example, Figure 2 shows a SubDomain for foo that says that when the sub-component bar is run, it can *also* have write permission to the /etc/otherwrite file. Conversely, it says that when foo runs the sub-component baz, it may not write to the /etc/writeme file.

Sub-components may also want a SubDomain that is completely unrelated to the parent domain. For example, a web server application might need to send some e-mail while processing a web form, and thus invokes a mail delivery agent whose SubDomain is completely different. We support this need with *absolute subdomains*, denoted by a subdomain specification following an x without a + or a -. Figure 3 shows an example absolute subdomain in which the bar program run from the foo program has access to a completely different subdomain than the foo program.

```
foo {
  /etc/readme      r ,
  /etc/writeme     w ,
  /usr/bin/bar     x {
    /usr/lib/otherread r ,
    /var/opt/otherwrite w ,
  } ,
}
```

Figure 3: Absolute SubDomain.

When a confined process tries to perform a file operation that is not permitted, two things happen:

1. The syscall returns with the error EPERM, just as if the attempt had failed due to a standard UNIX file system permission error.
2. The kernel generates a syslog entry describing the attempted violation. Intrusion detection systems can thus collect what ever information they want, and act accordingly, e.g., kill the offending process if such drastic steps are desired.

Sub-process Confinement

The section on related work describes several other systems that provide program-confinement mechanisms. However, with the exception of Java [4] the smallest component that they can confine is a native OS process. In contrast, SubDomain provides the unique feature of being able to confine components that are only a portion of an OS process. Historically of little practical interest, the need for sub-process confinement comes from the rise in popularity of scriptable servers and loadable modules. Let us expand upon these concepts.

A "scriptable server" is a server program that, from time to time, interprets a script or a program within itself, i.e., server-side includes [5], PHP web pages [6], Java servlets [3], etc. Such scripts are legitimately sub-component programs requiring separate confinement. Scriptable servers often have their own security mechanisms, but in depending on such restrictions, we are depending on application correctness, which is the dependence we seek to avoid in the first place. We would rather have a confinement mechanism that can be enforced by the operating system so that we do not depend on the correctness of the server application.

"Loadable modules" or "plug-ins" is the notion of providing a (fairly) fixed API in an application so

that extensions to can be loaded into the application, either at start-time or run-time. “Plug-in” is the common term for desktop applications (i.e., Netscape Navigator & Shockwave, Microsoft Word and EndNote) while “module” is the common term for servers (e.g., Apache and mod_perl).

The mod_perl module for Apache provides a perfect example of the sub-process problem. PERL scripts run at the behest of the Apache web server are normally interpreted by starting a separate process to run the PERL interpreter, and then interpreting the PERL script in that separate process. mod_perl loads a PERL interpreter directly into the Apache process to avoid the cost of starting the PERL interpreter process. While this is good for server throughput, it is bad for security:

- Bugs in mod_perl can crash the Apache web server process.
- Program-confinement mechanisms that only operate on OS processes cannot confine scripts interpreted by mod_perl separate from the Apache web server process.

The SubDomain solution to the “scripting & module” problem is to provide for subdomains for sub-process components, in cooperation with the enclosing application. The notation for a sub-process subdomain is unchanged from that of separate-process subdomains shown in Figure 1 through Figure 3. The effect is to create a variety of “hats” that process can wear, one for each sub-process component that it calls. The “cooperation” required from the enclosing program is that it should call the new change_hat() system call before calling the sub-process component.

The requirement to call change_hat() implies that we are once again trusting the application, which SubDomain is supposed to avoid. However, we are trusting the application code a great deal *less*, in that the application only has to make appropriate calls to change_hat(), which is much simpler than constructing and enforcing an effective “sandbox” environment [20]. Successfully calling change_hat() with the name of a sub-component before calling the sub-component seems easy enough to do correctly.

In addition to the requirement that enclosing application correctly calls change_hat(), we also require that the sub-component does *not* call change_hat() to escape to a more liberal subdomain. Here, we employ a *cookie* argument to change_hat() to prevent the confined module from escaping. The containing process initially calls change_hat() with a particular cookie value, and further change_hat() calls that do not provide a matching cookie argument are treated as security violations.

Thus for the containing process to prevent sub-component from escaping from the change_hat() SubDomain, it need only provide a cookie value that the contained sub-component cannot easily guess. We recommend fetching a word from /dev/random, but any reasonable source of entropy can be used.

The security of this method depends on the sub-component *not* being able to read the parent process’s cookie value. Here, SubDomain can leverage the power of *language*-based security protection systems such as proof-carrying code [30], strong type checking [39, 26, 37], and other language-based protection schemes [24, 40]. Such methods can, in principle, *prove* that the sub-component will not invoke the change_hat() system call.

Programming language techniques provide powerful protection, but also impose significant practical constraints, not the least of which is that the sub-component needs to be written in a particular language. In practice, we can still get reasonable assurance that the sub-component cannot read the containing process’s cookie value if it is written in a scripting language, i.e., a language that is *interpreted* rather than one compiling to native CPU instructions. In the practical setting of scripts for web servers, most such programs that are executed by loadable modules are scripting languages, e.g., PERL [41], PHP [6], and Java [3]. While no *formal* assurances are available, in practice it is easy to trust, say, mod_perl to not address random memory.

To see the power of this approach, consider the chronic problem of securely supporting Microsoft’s “Front Page Extensions,” a collection of non-standard HTML tags that the server interprets to provide more dynamic HTML content. Microsoft provides a mod_fp Apache module and collection of helper programs, but they have a poor security history [36]. There is no current practical method to securely support mod_fp.

SubDomain can solve the mod_fp problem by treating web pages containing “Front Page Extensions” tags as sub-components, and assigning each such page to a subdomain. So long as the mod_fp module can be trusted not to call the change_hat() system call, then no errant action of mod_fp can violate the security policy of the subdomain for the page it is interpreting.

SubDomain Implementation

The basic architecture of SubDomain is shown in Figure 4. The SubDomain policy engine is implemented as a Linux [38] loadable kernel module. Following the usual UNIX permissions checking, the relevant system calls (open(), exec(), read(), etc.) are modified to check if the calling process is a confined process. If so, the request is referred to the SubDomain module for further inspection. The SubDomain module then either returns normally (if the request is permitted) or returns an EPERM error (if the request is denied).

Once loaded, the SubDomain module disables module unloading to prevent tampering with the SubDomain policy engine. A user-level parser reads subdomain profiles from /etc/subdomain.d/* to convert the textual representation of profiles into kernel data structures, and inserts the updated profiles into the

kernel via a `sysctl()` interface. By convention, the `/etc/subdomain.d/foo` file would confine the `foo` program, but as shown previously, the actual name of the confined program is in the file, so confining multiple components with a single file is possible. Only root processes can access this kernel interface, and SubDomain-confined programs may not *access* the profile interface. In future work we will add further authentication requirements to the kernel's profile interface.

SubDomain Parsimony

SubDomain is simpler than competing least privilege mechanisms described in the section on related work in both implementation and usage. With regard to implementation, the SubDomain module and kernel patches amount to 4500 lines of C code, and the non-kernel parser is 825 lines. In contrast, the DTE kernel enhancement [7, 8] is over 40,000 lines of code. The relatively simple semantics of SubDomain enable a smaller implementation. "Smaller" is important for security systems, where correctness is critical, because bugs are approximately linear in code size.

SubDomain's usage is simpler than its competitors in that it is easier to devise and inspect SubDomain confinement profile than in other systems, which we elaborate on in the next section.

SubDomain Compatibility

We test the compatibility of SubDomain by putting it to work confining a variety of software components common to Internet servers, both large and small. SubDomain can confine binary-only programs, so long as there is no need for sub-process confinement. If sub-process confinement is required, then the program source needs to be edited and re-compiled to insert appropriate calls to `change_hat()`.

Like the "synthetic user" approach mentioned earlier, SubDomain confinement requires administrator intervention. However, SubDomain confinement is easier for the administrator in the following ways:

Ease of Application: A SubDomain profile does not interfere with any other aspects of the system except the SubDomain mechanism. Thus it

is easy to install a SubDomain profile along with the confined program. In particular, because the SubDomain profile is independent of the system the program is being installed on, the profile can be *included* with the program being distributed. In contrast, it is difficult to include a synthetic user in conventional program packages (e.g., tar balls or RPM packages).²

- It is easy for the administrator to inspect a SubDomain specification to determine the precise aspects of the system that are exposed to that program. In contrast, the exposure entailed by a synthetic user is non-obvious: the administrator must consider all files that are accessible to "anybody," which is a non-trivial exercise on non-trivial file systems.

The Kernel Wrapper approach [23] provides for confinement scripts that are full Turing-equivalent programs. While this provides extensive flexibility, it also means that the completeness and safety of an inserted kernel wrapper is not amenable to automatic analysis. In contrast, SubDomain profiles are easy to inspect to determine the security implications of updating a SubDomain profile. Furthermore, it is *strictly* safe to install a SubDomain profile where none existed before, because SubDomain strictly limits program privileges.

These factors have important implications for software distribution. Because SubDomain confinement profiles are system independent and guaranteed to be safe to install, it becomes feasible to package SubDomain confinement *with the program itself*. Thus an end user can consider installing a new program on a server appliance, and because of the SubDomain confinement information packaged with the program, the user can understand the security implications of installing that program. In future work, we plan to

²Note that bundling synthetic user IDs is exactly the approach taken by `qmail` [11], which results in excellent security for `qmail`, but also imposes substantial packaging difficulties that have hampered `qmail`'s spread.

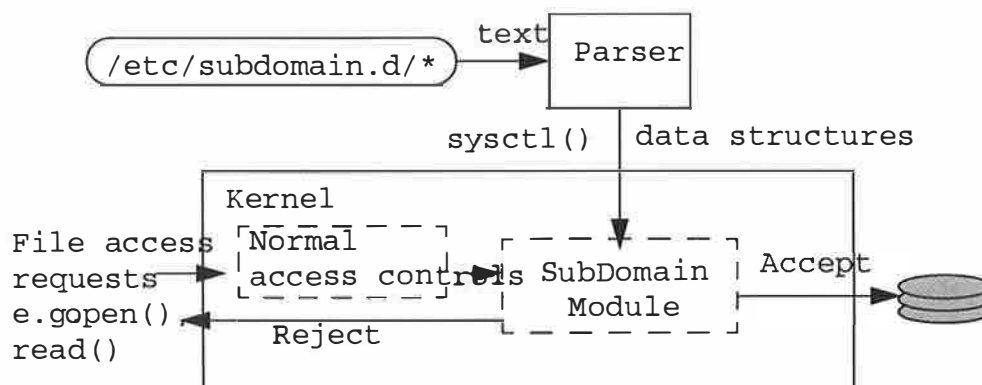


Figure 4: SubDomain implementation

develop future tools that will assist the administrator in determining the security implications of a set of SubDomain confinements

Which programs need to be confined with SubDomain depends on the convenience and security needs of the host system, and thus is an adjustable policy. The administrator can specify which of the following classes of programs must be confined with SubDomain before they are allowed to execute at all:

All Programs: All programs that execute on the host must be associated with a SubDomain, either explicitly, or inherited from a SubDomained parent program. This mode is suitable for bastion hosts.

All Listed User-IDs: All programs running under one of the user-IDs specified by the administrator must be associated with a SubDomain. For instance, the httpd user-ID runs many programs on behalf of the web server, and SubDomain confinement ensures that these programs will not affect other parts of the system. This mode is suitable for confining a potentially vulnerable collection of services on a system that also hosts critical data.

All root Programs: All programs running with a real or effective user-ID of "root." This mode allows a SubDomain profile to be used to achieve the classic goal of breaking up root's all-too-powerful privileges. The (defunct) POSIX i.e. "capabilities" model subdivided root's powers into a static set of 32 separate groups of "capabilities", and individual programs could assume part of root's powers by flipping on one or more of these sets of capabilities. SubDomain allows arbitrary sets of privileges to be grouped together, rather than accepting the groupings specified by POSIX i.e.

Only Specified Programs: Only the programs that have a SubDomain specified are thus confined, i.e., "default allow." This mode assumes that all programs on the host are adequately secured *except* for the programs being SubDomained. While not especially secure, this mode is convenient, e.g., for use on a client workstation to run a suspect program recently downloaded from the Internet.

The procedure for confining a program is to start with a null subdomain specification, run the application, observe the system log for complaints about attempts to access files outside the subdomain, and then add those files to the subdomain specification. This procedure is presently manual, because due consideration is required for two stages in this procedure:

Running the application: The application needs to be run under all of the "kinds" of input that it is expected to experience in a production environment, i.e., a comprehensive test suite. Determining these inputs requires some knowledge

of the application to ensure complete coverage. Failure to provide complete coverage results in a subdomain that is too "tight", and the application will occasionally fail to access resources that it needs.

Granting the privilege: We are confining the application precisely because we do *not* trust it, and therefore we cannot automatically assume that every file the application tries to access under test is a legitimate file for the program to access. The file should be included in the subdomain only after due consideration of the security implications.

For applications where source code is available, predicting the set of required resources should be feasible. If anticipating the set of files an application needs to access is truly difficult, then it is quite likely that the application represents a significant security threat, and should not be installed on hosts requiring security.

For applications where source code is not available, a run-time testing methodology must be used to experimentally identify all of the file resources that a program may try to access. To facilitate this, we use the dep program that we developed for the InDependence project [16] (funded by a student grant from USENIX). This program uses strace() to monitor the execution of a subject program, and amasses a list of all the files accessed. dep's use of strace() imposes heavier performance and compatibility overhead than SubDomain, but is none the less sufficient for exploring the file system domain of many programs. To further ease use, dep accumulates files accessed across multiple runs, so that a large test suite can be applied, and then the list of files accessed inspected once at the end of testing.

```
/home/httpd/cgi-bin/Count.cgi {
/etc/ld.so.cache           r
/lib/lib*                  r
/lib/ld-linux.so.2        r
/etc/nsswitch.conf         r
/etc/wwwcounter.conf       r
/etc/localtime             r
/var/log/httpd/wwwcount/wwwcount_log rw
/var/lib/wwwcount/*        r
/var/lib/wwwcount/data/*   rw
}
```

Figure 5: SubDomain for wwwcount CGI script

An example subdomain profile is shown in Figure 5, providing all of the resources needed to run the wwwcount CGI program (a popular web page hit counter program). Note the use of simple globbing to reduce the size of the subdomain specification when access to an entire directory is required. Figure 6 shows a more elaborate profile for the Apache web server itself, under a particular configuration. A list of some of the programs that we have confined and tested, along with the size of their subdomains, are listed in Table 1.

```

/usr/local/apache/bin/httpd {
  /
  /dev/null
  /dev/urandom
  /etc/group
  /etc/hosts
  /etc/host.conf
  /etc/ld.so.cache
  /etc/localtime
  /etc/nsswitch.conf
  /etc/passwd
  /etc/resolv.conf
  /home/httpd/perl/*
  /lib/*
  /usr
  /usr/lib/gconv/ISO8859-1.so
  /usr/lib/gconv/gconv-modules
  /usr/lib/perl5/5.00503/*
  /usr/lib/perl5/site_perl/5.005/i386-linux/*
  /usr/local
  /usr/local/apache
  /usr/local/apache/conf/*
  /usr/local/apache/htdocs/*
  /usr/local/apache/logs*
  /usr/share/locale/en_US/*
  /usr/share/locale/locale.alias
}

```

Figure 6: SubDomain for Apache Web Server.

SubDomain Performance

Here we present a variety of SubDomain performance measurements. The next section describes our microbenchmarks on mediated system calls, and the section after that describes our macrobenchmarks on a confined PERL script interpreted by the mod_perl Apache module.

Microbenchmarks

Here we use the usual benchmarking technique to measure affected system calls by crafting programs that issue each system call 10,000 times, run the programs several times, discard the first run to avoid cold cache effects, and average the remainder. All tests were performed on a dual-processor Pentium III 700 MHz, with 256 MB of RAM. Table 2 summarizes these results. We include measurement of the `get_pid()`

System Call	Standard Cost	SubDomain Cost	% Overhead
<code>fork()</code>	295	295	0%
<code>exec()</code>	1387	1487	7%
<code>open()</code>	3.71	5.39	45%
<code>get_pid()</code> vs. <code>change_hat()</code>	1.81	4.70	159%

Table 2: SubDomain Microbenchmarks in microseconds.

```

/perl0/cgittest-001.cgi {
  /usr/lib/perl5/site_perl/5.005/i386-linux/Apache/Registry.pm
  /etc/localtime
  /usr/lib/perl5/5.00503/*
  /home/httpd/perl0/cgittest-001.cgi
  /home/httpd/perl0/cgitemplate-001.html
  /home/httpd/perl0/cgidata-001
  /var/log/httpd/*
}

```

Figure 7: Test PERL script's SubDomain profile.

system call as a baseline for comparison against the `change_hat()` system call, as `get_pid()` is commonly regarded as the simplest system call.

As expected, the major overhead appears in the `open()`, `exec()` and `change_hat()` system calls, where SubDomain is checking the action against the subdomain specification for the confined process.

Program	Size of SubDomain
Simple bash shell script	31 files
PHF CGI program	14 files
CGI Mail program	7 files
htsearch CGI program	11 files
wwwcount CGI program	10 files
Apache web server	33 files
lpd	16 files
lpq	10 files
lpc	11 files
Postfix Mail Delivery Agent	15 files
Postfix-script helper program	65 files

Table 1: SubDomain-confined programs.

Macrobenchmarks

Our macrobenchmark is SubDomain confinement of a PERL script to be executed via the mod_perl Apache module, thus exercising SubDomain's capability to confine *active content* scripts. To exercise the web server's cache, we replicated the PERL script 1000 times, and used the Webstone performance benchmark to measure the overhead cost of PERL scripted web pages protected with SubDomain vs. without protection. The PERL script itself reads two files with some busy-work in between, simulating a script that fetches a "container" template from one file, HTML content from another file, and does some interim processing to merge the two, e.g., compute a

hit counter. The SubDomain profile for this script is shown in Figure 7.

The test environment used the same dual-processor Pentium III 700 MHz server with 256 MB of RAM, and a private network (crossover cable) via 100 Mbit ethernet.

The test results are shown in Table 3, measured for 5 to 10 concurrent client connections. Tests were run twice, and the results averaged. For all cases, the SubDomain overhead is between 1% and 2%, i.e., in the noise range.

Related Work

Here we describe work that, similar to SubDomain, specifically attacks the problem of confining suspect programs. Despite the age of the notion of least privilege [33], much of this work has emerged relatively recently. It is our conjecture that this is a result of a shift in emphasis from defending secrecy (the dominant concern for military organizations) to defending *integrity* (the dominant concern for Internet-connected businesses) and the emergence of the notion of *survivability* [35]. This list of related work is necessarily partial, as the total body of related work is very large.

TRON

The TRON system [10] is a kernel enhancement for ULTRIX that can confine a program's execution to

a protection domain consisting of a finite set of *capabilities* in the form of file names. TRON adds the `tron_fork()` system call, which functions exactly like the classic `fork()` system call, except that it specifies the protection domain as an extra argument. TRON is semantically most similar to SubDomain: the protection domains are the same (sets of files) and are similarly applied to host programs, orthogonal to user privileges. The major differences are:

- TRON is discretionary, while SubDomain is mandatory. TRON provides user commands to run programs in a confined domain, while SubDomain always runs a specified program in a confined domain. Thus in the usual DAC vs. MAC trade-off, TRON is more convenient for individual users, while SubDomain is more convenient for securing entire systems, e.g., server appliances.
- TRON's finest granularity is the ULTRIX process; it cannot confine loadable modules.

Janus

Janus [25] is a user-level mechanism for confining programs to a specific set of resources. Intended to confine "helper" applications run from within a Web browser, Janus uses the `ptrace()` system call and a monitoring process to mediate all system calls made by the helper application. If the action proposed by the helper application violates a policy set by the user, then the monitoring process rejects the request. This

Test	# of Clients	Connection Rate	Avg. Response Time (ms)	Avg. Client Throughput
Standard	5	75.97	66.5	26.29
SubDomain	5	75.19	66.5	26.02
% Overhead		1%	0%	1%
Standard	6	78.14	77	27.04
SubDomain	6	76.56	78	26.49
% Overhead		2%	1.3%	2%
Standard	7	78.38	89	27.13
SubDomain	7	77.24	90.5	26.73
% Overhead		1.45%	1.7%	1.5%
Standard	8	78.26	102	27.08
SubDomain	8	76.71	104	26.54
% Overhead		2%	2%	2%
Standard	9	78.24	115	27.08
SubDomain	9	77.02	116.5	26.66
% Overhead		1.6%	1.3%	1.6%
Standard	10	78.43	127	27.15
SubDomain	10	77.07	129.5	26.67
% Overhead		1.7%	2%	1.7%

Table 3: SubDomain macrobenchmarks with WebStone.

approach requires four system calls to be executed to effect one confined system call.

Java 2 Security

The Java 2 security model [4] allows the JVM to be configured to assign particular capabilities to designated Java *classes*, similar to the SubDomain notion of assigning file system capabilities to programs. This is an enrichment over the original Java security model [26] which assigned one fixed set of capabilities to remotely-loaded applets (almost nothing), and another fixed set of capabilities to locally-loaded applets (almost everything).

The Java 2 security mechanism is notable as the only system other than SubDomain capable of confining sub-process components, in that Java classes are typically smaller than the host OS processes. Naturally, the Java 2 security model does not apply to non-Java native executables.

chroot Jail

The `chroot()` system call³ makes the argument directory be the effective root directory, i.e., “/” for the invoking process. The point of this operation is that the file system domain for the affected process is now limited to the contents of the argument directory. Any files that the application needs to access must be placed inside the `chroot` directory, or the access will fail.

The `chroot` technique is a popular form of confinement, in large part because standard kernels (e.g., Linux) support it. However, `chroot` has defects in all three of the dimensions a security enhancement should address:

Security: `chroot` jails are resistant to oblivious attempts to escape the jail, i.e., attempts to access files that are not accessible within the jail. However, if the attacker can execute their own code within the `chroot` jail, it is fairly easy to break the jail and access outside files. Thus jailed programs generally cannot be trusted with strong privileges, i.e., it is insecure to depend on `chroot` to confine a root process.

Compatibility: Each `chroot`d program must have the necessary components of the file system replicated within its jail, which is problematic if the program requires access to a large, complex set of files, i.e., shell scripts need all invoked programs replicated into the `chroot` jail. Thus setting up a `chroot` jail can be a lot of tedious, complex work. The `chroot` technique also breaks programs that need to interact with other parts of the system.

Performance: Because `chroot` jails require duplication of all resources needed by the jailed program (soft or hard links could be used as escape routes) they consume excessive disk space and file system buffer cache space.

³“man `chroot`” on most UNIX Systems

Type Enforcement

The type enforcement work [13, 7, 8] has recently been extended to provide better support for program confinement. *Kernel hypervisors* [28] provide a facility for installing small state machines that intercept kernel system calls and enforce a security policy. Such a facility can be viewed as a tool that could be used to build a SubDomain-like least privilege system. Fraiser, Badger and Feldman provide a similar tool for building security policy enforcement automata [23]. SubDomain provides the following key advantages over this technique:

Parsimony: SubDomain is much simpler than the TE and DTE implementations; the SubDomain kernel code is approximately 1/10 the size of the DTE kernel patch. Simplicity is critical in security systems.

Safety: The DTE Wrapper system [23] allows code to be inserted into the operating system to perform mediation. While this is a powerful technique, it is also dangerous: *malicious* DTE wrapper code could just as easily be inserted. In contrast, SubDomain profiles are easy to inspect to determine the security implications of updating a SubDomain profile. Furthermore, it is *strictly* safe to install a SubDomain profile where none existed before, because SubDomain strictly limits program privileges.⁴

Application-Specific Mechanisms

Various application environments provide their own least privilege-like mechanisms. For instance, the PERL interpreter includes a facility known as “taint”, in which input provided to the PERL script *cannot* be used to formulate an action (i.e., `system()` operation) unless it has been “adequately” inspected by the PERL script [41]. PERL also includes a “safe PERL” facility, where in the programmer can specify a set of PERL operators that the script may not use.

Another application-specific least privilege mechanism is the notion of “wrappers.” For example, CGI Wrappers [31] causes a CGI script to be run with the user-ID of the script owner, rather than the user-ID of the web server. Combined with the synthetic user-ID notion described previously, CGI Wrappers can construct a least privilege environment for CGI scripts.

PACLs: Program-based Access Control Lists

We believe PACLs [42] to be the first instance of an access control system based on the program performing the operation. The PACL system is the exact dual of the SubDomain notion: files have an access control list that enumerates programs that are permitted to operate on that file. A simulated PACL system was built and evaluated, but an actual PACL system was never finished.

⁴This observation due to Blaine Bumham.

Status & Availability

The implementation is not complete with respect to the description in this paper.

- The absolute and relative sub-domains described earlier are not complete: child processes either inherit the parent's profile, or use their own profile if one is specified.
- The multiple modes of requiring SubDomain confinement described previously is only partially implemented. The implementation currently supports "paranoid" mode where all processes must have SubDomain confinement, and "open" mode, where only the programs that are specified are confined by SubDomain.

SubDomain is implemented for Linux, and is available from <http://immunix.org>. The kernel enhancement portion is licensed under the GPL, and the non-kernel portions are proprietary to WireX but available for free for non-commercial use.

Conclusions

Vulnerable software is a major security problem, mandating constant system administrator attention to keep systems up to date with vendor-supplied security patches. This is especially problematic for complex Internet servers, which are required to provide extensive services to anyone on the Internet. Some form of confinement mechanism to approximate least privilege is the generic solution, but often imposes more costs than administrators deploying in "internet time" can bear. Our SubDomain confinement mechanism advances over previous confinement work, simplifying both implementation and administration overheads by confining *programs* instead of *users*.

This approach enables SubDomain confinement to be packaged with programs, in contrast with confinement mechanisms that are bound to the system. SubDomain also provides fine-grained protection, confining software components *finer* than the host OS process, providing the unique capability to protect potentially vulnerable server *modules* such as Microsoft's Front Page Extensions to the Apache web server. We have implemented and tested the system, showing that it provides all three essential properties of a security enhancement: enhanced security, software compatibility, and preserved performance.

Author Information

Dr. Crispin Cowan is co-founder and Chief Research Scientist of WireX Communications, Inc., and is a Research Assistant Professor at the Oregon Graduate Institute, where he teaches graduate courses in system security. His research focuses on making existing systems more secure without breaking compatibility or compromising performance. Professor Cowan has authored 28 refereed publications, including those describing the StackGuard compiler for defending against buffer overflow attacks. Reach him electronically at crispin@wirex.com.

Steve Beattie is one of the original developers and the current maintainer of the StackGuard compiler enhancement. He is employed in the Advanced R&D Group at WireX Communications, Inc., who graciously pays him to work on such fun projects as StackGuard and SubDomain. He received a Masters Degree in Computer Science from the Oregon Graduate Institute, and was previously employed as a Jack-of-all-Trades Sysadmin in a Large Dead-Tree Publishing Corporation.

Calton Pu received his Ph.D. from University of Washington in 1986 and served on the faculty of Columbia University and Oregon Graduate Institute. Currently, he is holding the position of Professor and John P. Imlay, Jr. Chair in Software at the College of Computing, Georgia Institute of Technology. He is leading the Infosphere project that combines his research interests. First, he has been working on next-generation operating system kernels to achieve high performance, adaptiveness, security, and modularity, using program specialization, software feedback, and domain-specific languages. This area has included projects such as Synthetix, Immunix, Microlanguages, and Microfeedback, applied to distributed multimedia and system survivability. Second, he has been working on new data and transaction management by extending database technology. This area has included projects such as Epsilon Serializability, Reflective Transaction Framework, and Continual Queries over the Internet. He has published more than 30 journal papers and book chapters, 100 conference and refereed workshop papers, and served on more than 40 program committees. He is currently an associate editor of IEEE TKDE, DAPD, and IJODL.

Greg Kroah-Hartman is one of the main Linux USB developers, and the current Linux USB Serial and USB Bluetooth driver maintainer. He is also the author and maintainer of the Linux `usbview` program which is being shipped in most of the major Linux distributions. His free software is being used by more people than any closed source projects he has ever been paid to develop. He is currently employed in the Advanced R&D group at WireX Communications Inc, and has a Bachelors Degree in Computer Science.

Perry Wagle received his MS in Computer Science at Indiana University in 1995, and in 1997 he headed to the Oregon Graduate Institute to join the Immunix project's survivability research, where among other things, he was the primary programmer of the StackGuard enhancement to GCC. Rather than go with WireX like the rest of Immunix last year, he stayed at OGI to work on the Infosphere project. He is interested in applying programming language technology to operating systems problems. For example, survivability presents an interesting problem: how effectively can you transform or assist legacy code that is intolerant to security faults so that it responds sensibly to attacks?

Virgil D. Gligor received his B.Sc., M.Sc., and Ph.D. degrees from the University of California at Berkeley. He has been at the University of Maryland since 1976, and is currently a Professor of Electrical and Computer Engineering. He has worked in the areas of access control on several UNIX systems and is the co-designer of two automated tools, one for covert-channel analysis of (C language) source code (written in PROLOG), and the other for penetration analysis, which have been used by IBM Corporation for analysis of Secure Xenix and for TIS Inc Trusted Xenix. His work helped define precisely the notion of the denial of service, and received the Best Research Paper Award at the 1988 IEEE Symposium on Research in Security and Privacy for research work in this area (paper co-authored with his graduate student C.F. Yu). He is the co-author of several US patents in the areas of intrusion detection, penetration analysis methods and tools, and role-based access control.

References

- [1] Alpern, Bowen and Fred B. Schneider, "Defining Liveness," *Information Processing Letters*, 21(4):181-185, 1985.
- [2] Amoroso, Edward, *Fundamentals of Computer Security Technology*, Prentice Hall, Englewood Cliffs, NJ, 1994.
- [3] Anonymous, *The Java Web Server Architecture Overview*, <http://www.javasoft.com/products/java-server/documentation/webserver1.1/>, 1997.
- [4] Anonymous, *JDK 1.2 Security*, <http://java.sun.com/products/jdk/1.2/docs/guide/security/index.html>, March 1998.
- [5] Assorted, *NCSA HTTPd Tutorial: Server Side Includes*, <http://hoohoo.ncsa.uiuc.edu/docs/tutorials/includes.html>.
- [6] Assorted, *PHP Hypertext Processor*, <http://php3.org/>.
- [7] L. Badger, D. F. Sterne, et al., "Practical Domain and Type Enforcement for UNIX," *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, May 1995.
- [8] Lee Badger, Daniel F. Sterne, David L. Sherman, Kenneth M. Walker, and Sheila A. Haghighat, "A Domain and Type Enforcement UNIX Prototype," *Proceedings of the USENIX Security Conference*, 1995.
- [9] Brian Behlendorf, Roy T. Fielding, Rob Hartill, David Robinson, Cliff Skolnick, Randy Terbush, Robert S. Thau, and Andrew Wilson, *Apache HTTP Server Project*, <http://www.apache.org>.
- [10] Andrew Berman, Virgil Bourassa, and Erik Selberg, "TRON: Process-Specific File Protection for the UNIX Operating System," *Proceedings of the 1995 Winter USENIX Conference*, USENIX Association, 1995.
- [11] D. J. Bernstein, *qmail*, <http://cr.yp.to/qmail.html>, 1990.
- [12] M. Bishop and M. Digler, "Checking for Race Conditions in File Accesses," *Computing Systems*, 9(2):131-152, <http://olympus.cs.ucdavis.edu/bishop/scriv/index.html>, Spring 1996.
- [13] W. E. Bobert and R. Y. Kain, "A Practical Alternative to Hierarchical Integrity Policies," *Proceedings of the 8th National Computer Security Conference*, Gaithersburg, MD, 1985.
- [14] CERT, *Advisory CA-96.06: Vulnerability in NCSA/Apache CGI Example Code*, ftp://info.cert.org/pub/cert_advisories/CA-96.06.cgi_example_code, September 1996.
- [15] CERT, *Advisory CA-98.05: Multiple Vulnerabilities in BIND*, ftp://info.cert.org/pub/cert_advisories/CA-98.05.bind_problems, May 1998.
- [16] Crispin Cowan, Ryan Finnin Day, and Hao Zhao, *InDependence: Automating the Discovery of Application Dependencies*, <http://www.cse.ogi.edu/DISC/projects/independence>, 1997.
- [17] Crispin Cowan, Calton Pu, Dave Maier, Heather Hinton, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang, "StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks," *7th USENIX Security Conference*, pages 63-77, San Antonio, TX, January 1998.
- [18] Crispin Cowan, Perry Wagle, Calton Pu, Steve Beattie, and Jonathan Walpole, "Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade," *DARPA Information Survivability Conference and Expo (DISCEX)*, January 2000; Also presented as an invited talk at SANS 2000, Orlando, FL, <http://schafercorp-ballston.com/disceX>, March 23-26, 2000.
- [19] Michele Crabb, "Curmudgeon's Executive Summary," *The SANS Network Security Digest*, Michele Crabb, editor, Contributing Editors: Matt Bishop, Gene Spafford, Steve Bellovin, Gene Schultz, Rob Kolstad, Marcus Ranum, Dorothy Denning, Dan Geer, Peter Neumann, Peter Galvin, David Harley, Jean Chouanard, SANS, 1997.
- [20] Drew Dean, Edward W. Felten, and Dan S. Wallach, "Java Security: From HotJava to Netscape and Beyond," *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, <http://www.cs.princeton.edu/sip/pub/secure96.html>, 1996.
- [21] eEye, *IIS Remote Hole*, <http://www.eye.com/database/advisories/ad06081999/ad06081999.html>, June 1999.
- [22] David F. Ferraiolo and Richard Kuhn, "Role-Based Access Control," *Proceedings of the 15th National Computer Security Conference*, Baltimore, MD, October 1992.
- [23] Tim Fraser, Lee Badger, and Mark Feldman, "Hardening COTS Software with Generic Software Wrappers," *Proceedings of the IEEE*

- Symposium on Security and Privacy*, Oakland, CA, May 1999.
- [24] Neal Glew and Greg Morrisett, "Type-Safe Linking and Modular Assembly Language," *Twenty-Sixth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 250-261, San Antonio, TX, <http://www.cs.cornell.edu/talc/papers.html>, January 1999.
 - [25] Ian Goldberg, David Wagner, Randi Thomas, and Eric Brewer, "A Secure Environment for Untrusted Helper Applications," *6th USENIX Security Conference*, San Jose, CA, July 1996.
 - [26] James Gosling and Henry McGilton, "The Java Language Environment: A White Paper," <http://www.javasoft.com/docs/white/langenv/>, May 1996.
 - [27] J. A. McLean, "A General Theory of the Composition for Trace Sets Closed Under Selective Interleaving Functions," *Proceedings of the IEEE Symposium on Security and Privacy*, pages 79-93, Oakland, CA, May 1994.
 - [28] Terrance Mitchem, Raymond Lu, and Richard O'Brien, "Using Kernel Hypervisors to Secure Applications," *Proceedings of the Annual Computer Security Application Conference*, December 1997.
 - [29] Mudge, *How to Write Buffer Overflows*, <http://l0pht.com/advisories/bufero.html>, 1997.
 - [30] George C. Necula and Peter Lee, "Safe Kernel Extensions Without Run-Time Checking," *Proceedings of the USENIX 2nd Symposium on OS Design and Implementation (OSDI'96)*, <http://www.usenix.org/publications/library/proceedings/osdi96/necula.html>, 1996.
 - [31] Nathan Neulinger, *CGIWrap: User CGI Access*, <http://www.unixtools.org/cgiwrap/>, 1997.
 - [32] Aleph One, "Smashing The Stack For Fun And Profit," *Phrack*, 7(49), November 1996.
 - [33] Jerome H. Saltzer and Michael D. Schroeder, "The Protection of Information in Computer Systems," *Proceedings of the IEEE*, 63(9), November 1975.
 - [34] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman, "Role Based Access Control Models," *IEEE Computer*, pages 38-47, February 1996.
 - [35] Howie Shrobe, *ARPATech '96 Information Survivability Briefing*, http://www.darpa.mil/ito/ARPATech96_Briefs/survivability/survive_brief.html, May 1996.
 - [36] Marc Slemko, *Microsoft FrontPage 98 Security Hell*, <http://www.worldgate.com/marcs/fp/>, October 1997.
 - [37] Robert E. Strom and Shaula Alexander Yemini, "Typestate: A Programming Language Concept for Enhancing Software Reliability," *IEEE Transactions on Software Engineering*, 12(1):157-171, January 1986.
 - [38] Linus Torvalds, et al., *Linux Operating System*, <http://www.linux.org/>.
 - [39] United States Department of Defense, *Reference Manual for the Ada Programming Language ANSI/MIL-STD-1815A-1983*, United States Department of Defense, February 1983.
 - [40] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham, "Efficient Software-Based Fault Isolation," *Proceedings of the Fourteenth ACM Symposium on Operating System Principles (SOSP'93)*, pages 203-216, Asheville, NC, December 1993.
 - [41] Larry Wall, Tom Christiansen, and Randal L. Schwartz, *Programming Perl*, O'Reilly & Associates, Inc., 2nd edition, 1996.
 - [42] D. R. Wichers, D. M. Cook, R. A. Olsson, J. Crossley, P. Kerchen, K. Levitt, and R. Lo, "PACL's: An Access Control List Approach to Anti-viral Security," *Proceedings of the 13th National Computer Security Conference*, pages 340-349, Washington, DC, October 1-4 1990.

NOOSE – Networked Object-Oriented Security Examiner

Bruce Barnett – General Electric Corporate Research & Development

ABSTRACT

NOOSE (Networked Object-Oriented Security Examiner) is a distributed vulnerability analysis system based on object modeling. It merges the functionality of host-based and network-based scanners, storing the results into several object classes. The remote agents are implemented as dynamically extended PERL agents. NOOSE is able to collect vulnerabilities from a variety of sources, including outputs from other vulnerability analysis programs (e.g., Muffet's CRACK), collecting information from systems that may or may not have cooperative agents on them. Communication is based on a secure, reliable datagram protocol implemented as a set of PERL object classes. Unlike some vulnerability systems, NOOSE presents the vulnerability information as an integrated database, showing how vulnerabilities may be combined into chains across multiple accounts and systems. It understands unconditional vulnerabilities (i.e., stack-overflow, password guessing) along with conditional (Trojan horse, rlogin, and NFS access). Conditional vulnerabilities gain limited or privileges if conditions exist, such as access to specific accounts. The information is presented as an object-oriented "spreadsheet" format, allowing the security manager to explore vulnerabilities at whim. Once complete, the vulnerability analysis can move both forwards and backwards interactively, showing both what a selected account can attack, as well as showing who can attack a selected account. Besides vulnerability analysis, the system can intelligently verify the installation of security patches, dynamically installing missing patches. NOOSE is therefore a flexible prototype, able to provide a subset of the functionality of COPS, SATAN and TRIPWIRE, yet because of the object model, be used for developing new paradigms, such as reacting to intrusions, information warfare, and survivability management systems.

Problem Statement

This paper discusses limitations in *Vulnerability Analysis systems* such as COPS, SATAN Tiger, RSS and ISS. For convenience, these systems will be referred to as *VA systems*. In this paper, a vulnerability is a potential path to break into someone's account to elevate their privilege. This paper also discusses *vulnerability chains*, which is defined to be two or more vulnerabilities, that can be executed in sequence. An example of a chain is using NFS to insert a Trojan horse into a directory, which can be executed by a system administrator, to gain root access to a file server. Once this key account has been breached, the group of related clients become vulnerable because of the relationship between servers and clients. This collection of systems will be called a *workgroup*. In a large facility, there may be dozens or hundreds of workgroups, with separate administrators.

The goal of this paper is to find a way to identify and eliminate these vulnerability chains among and between workgroups. The author believes current VA systems have difficulty in doing this because of the following reasons:

- Some potential vulnerabilities are accepted as a matter of policy, and the convenience value overrides the potential risk. These are considered conditional vulnerabilities, because other conditions determine how risky it is.

- Vulnerability chains composed of one or more conditional vulnerabilities across multiple systems aren't evaluated.
- The results of host-based scanning is not integrated with the results of network-based scanning.
- Some systems, such as name servers or file servers, need stronger protection than clients. Likewise, some accounts, such as system administrators, are key elements in protecting the overall security. However, vulnerability analysis systems ignore these different threat potentials.
- Many security systems characterize potential failures with a simplistic red/yellow/green indicator or a numbered scale with 5 values. This is too coarse a measurement to be useful. The simple statement "NFS is insecure" is often ignored, while a report that identifies the precise directory that can be used to compromise the root account is more likely to be fixed.

The author feels that many current VA systems don't properly analyze the consequences of an intruder who breaches a firewall, or an insider attempting to gain access. Systems have complex relationships between clients and servers, and the compromise of a single server can allow hundreds of clients to be compromised. Servers can also be clients of other servers, and clients can be used to compromise other servers.

Likewise, accounts have complex inter-relationships, and system administrator accounts require special protection.

The vulnerability chain that was mentioned earlier, for instance, makes it trivial to break into an account without detection. Using low-level NFS calls (e.g., Leendert van Doorn NFSSHELL) typically requires no special privileges and allows access to any file on an NFS-exported file system not owned by root, such as those owned by a system administrator. If that account is not directly accessible, a Trojan horse can often be used. Since these attack mechanisms use ordinary file access mechanisms, they are rarely detected. Therefore it is essential that the potential danger be reported and repaired. Identifying this type of danger is one of the primary goals of NOOSE.

There have been attempts to integrate information, but these have been limited to a common output format [12], or a common user interface [11]. Kuang [1] identified vulnerability chains, but was limited to a single system. NetKuang [5] is one attempt to correct this. NOOSE uses a second approach.

Background

This implementation was based on our experience with an earlier Expert Fault Manager system [13], where the importance of relationships was emphasized. The author decided to apply what was learned while measuring security risks. However, several custom agents were on each system, and they needed to be upgraded manually. The communication system suffered from deadlocks occasionally. The core components were enhanced and applied to a vulnerability analysis prototype [14]. The work was influenced by the author's program for analyzing Trojan horses on UNIX systems, and various tools from Purdue [2, 3, 4]. This project was started in January of 1996, with funding was provided by Lockheed Martin and L-3 Communications. This paper describes the lessons learned from this prototype.

The author choose the name NOOSE because of the concept of drawing a circle around a set of arbitrary systems, and identifying vulnerabilities within this set, using OO techniques.

Description of System

The NOOSE system is written using PERL, chosen because of the power of the language, string parsing, and the ease of prototyping, as well as being able to potentially migrate code from COPS and SATAN/SAINT variants. NOOSE is implemented with 26 PERL-based object classes. A PERL agent exists on all systems cooperating with NOOSE. These agents talk to NOOSE using a centralized dispatcher, which in turn talks to an Information Warfare (IW) module. A Graphic User Interface (GUI) is written in PERL and TK, and provides a spreadsheet-like interface to the information contained in the IW module. The overall architecture can be seen in Figure 1.

Auxiliary files exist that contain information about patches, operating systems, and PERL modules to be uploaded to the agent.

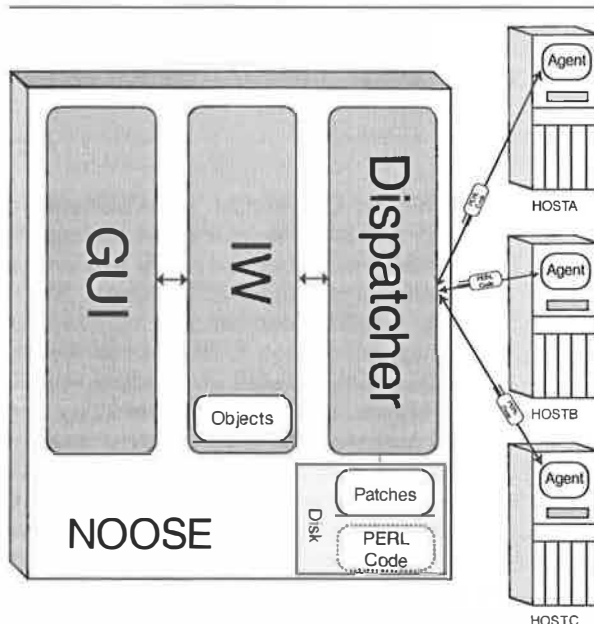


Figure 1: NOOSE architecture.

The agents just contains enough information to listen for new commands. New subroutines can be uploaded, and the revision tracked. If modules require other modules that are undefined, this is reported as an error. Therefore the agents are small, easy to install, and never need to be updated manually, which reduces maintenance costs. A simple dependency system was put in place, to identify required modules. However, the implementation used a single file for each operating system variant, and uploaded the entire file to the agent on demand. Therefore the dependency feature was rarely used.

The agent will run on a Windows NT box, but no vulnerabilities are currently gathered.

Object Classes

The primary goal of object-oriented programming is the ability to reuse object classes, lowering cost of development. Therefore our goals was to develop an object model that can support multiple algorithms. A data structure that can be used with two diverse algorithms has a better chance of being used by future algorithms. Object modeling provides a concise mechanism to document the data structures used in a system, often on a single page.

Object Model

The implementation contains of the following object classes:

- Communication (9 classes)
- System-related (1 Primary base class, 1 secondary base class, 13 sub-classes)
- GUI (2 classes)

System-related object classes

The NOOSE system uses the following object model, as shown in Figure 2. The 13 different object classes maintain information about remote systems, corresponding to their state:

- Patch
- Host
- Signature
- Operating System
- Account
- UID
- Vulnerability
- UNIX Resource (which has four sub classes)
 - File
 - Link
 - Directory
 - Missing
- FileState
- PatchState

All system-related objects have a unique name. NOOSE uses a simple ASCII string composed of the following parts:

Object Class

Host responsible for object
Object-specific information

Examples are

```
account/pluto/smith
uid/pluto/214
host/pluto
dir/pluto/etc/mail/sendmail.cf
```

This provides a simple, extensible way to create new object types, as well as a simple method of locating the authority of the object (in this case, the host that must be queried to get information about that object.) In our communication paradigm, *instances* of objects resided on different hosts.

Relationship Superclass

All of the 13 system-related classes are derived from a base class that provides object lookup, creation, deletion, as well as relationship creation, querying, and navigation. This relationship or association is critical to the implementation. It creates one-to-one, one-to-many, and many-to-many relationships between

NOOSE Object Model (OMT Notation)

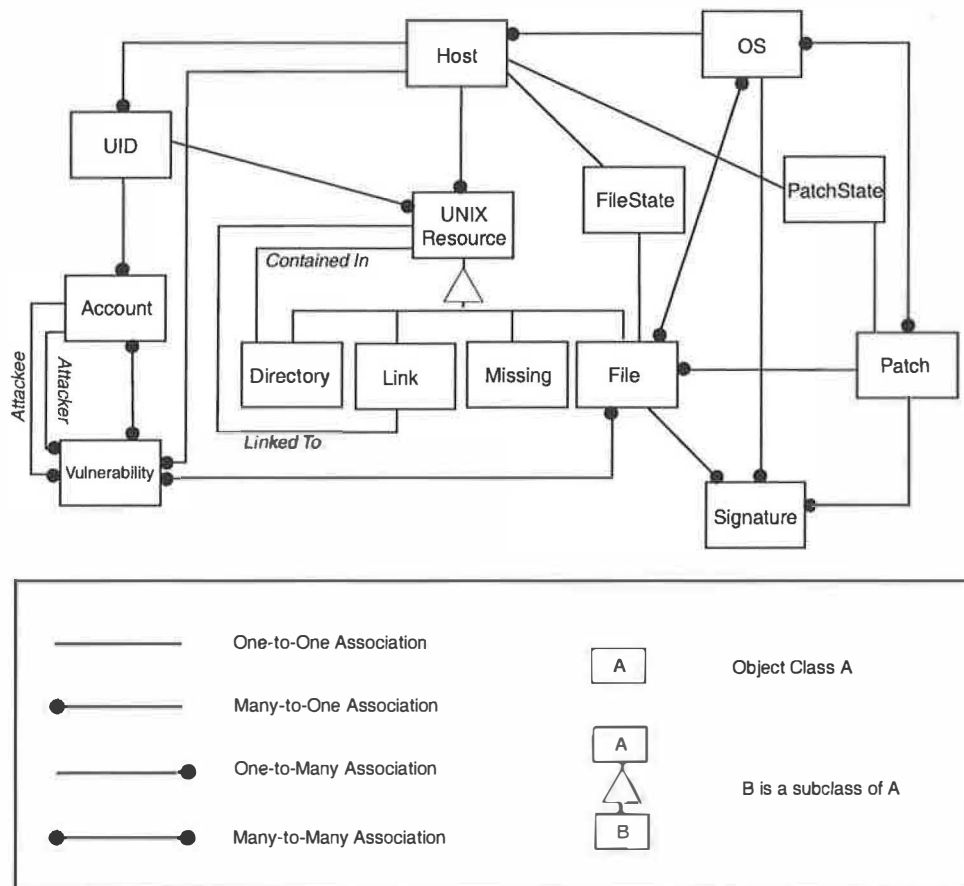


Figure 2: NOOSE object model.

any two system-related classes. Creating a one-to-many relationship between two instances (e.g., there are several files related to a patch) merely requires associating two object references with the method call:

```
$patchObject->one_to_many($fileObject);
```

No additional declarations are necessary. This simplified modifications of the object model. Other methods are *one_to_one*, *many_to_one* and *many_to_many*. If a one-to-one relationship is created, and a second relationship to the same class is added, an error is generated, suggesting a one-to-many relationship be used instead. The base class also provides ways to obtain, find, test, search and integrate related objects. It can be used as a collector object, and simplifies algorithm development significantly. The code fragment below demonstrates the methods as it finds and tests all of the required patched files corresponding to a revision of an operating system; see Listing 1.

These system-related classes, besides used to store information about the objects, allow algorithms to be easily constructed based on the relationship (or association) between objects. Often the relationship between two classes need not have a specific name, and the relationship is obvious from the context. Some objects have very few attributes, as the primary purpose is collection and navigation.

Two key classes are Accounts and Vulnerabilities. An account object is a username/hostname pair. The vulnerability object always has relationships to two accounts (except when the account has been removed, in which case it refers to the UID object class, which corresponds to the user ID number. In our system, a vulnerability is a potential mechanism to allow someone to go from one account to a different account.

The first relationship is to the account that can be compromised (the *attacker*), while the second shows the account that can compromise the first account (the

attacker). Vulnerability objects may have an optional relationship to a file or directory, indicating the cause of the vulnerability.

Wildcards in Account names

Accounts consisted of two pieces of information: the username, and the hostname. NOOSE uses a "*" to indicate a wildcard, which may be in either the host, username or both. Accounts with wildcards are used to describe specific vulnerability classes. If an account had no password, then anyone on any system could access that account. This fictitious user is indicated by the account "account/*/*." If Joe Smith's account has a "+" in the ".rhosts" file, then the vulnerability can be initiated from the account "account/*smith". If Smith's account on host "pluto" has a "." first in the searchpath, then this account can be compromised by the "account/pluto/*" account, which means anyone on host *pluto*. This simple naming convention can be used to describe the starting attack point of any vulnerability. In the case of vulnerabilities within a group or netgroup privileges, multiple vulnerabilities are created, with the attacking accounts expanded to the complete list of individuals within this group.

Vulnerability Chains

A vulnerability chain occurs when multiple vulnerabilities can be used to achieve a particular goal (or in this case, an account). Figure 3 shows such a chain.

In this case, assume a hacker can break into the *lpd* account on host *hosta* because it was missing a security patch. Next the attacker uses *NFSSHELL* to access the home directory of account *hostb/smith*. This account has write privileges in */usr/local/bin* and a Trojan horse is created. The *backup* account has this directory in the searchpath, and executing the Trojan horse compromises the account. Once done, the user may gain access to the *root* account on a file server, which allows access to all of the clients.

```
#Specify host to check
$os = Os->fetch($host->get_os_type); # Find the OS type and revision
foreach $patch ($os->get_many("Patch")) { # get the patches for the OS
    # Get the files included in each patch cluster
    foreach $file ($patch->get_many("File")) {
        # Files have more than one signature - depends on OS rev
        foreach $signature ($file->get_many("Signature")) {
            # Only look at those that match the OS and revision
            if ($signature->has($os)) { found a match
                $signature->verify(
                    host=>$host,
                    file=>$file,
                    patch=>$patch);
            }
        } # Signatures
    } # files
} # Patches
```

Listing 1: Testing patched files.

Communication classes

All network communication is comprised of objects and methods, which are specified as ASCII strings. The dispatcher looks at the object, determines the host to send the message to, and sends the information to the clients using a protocol layered on top of

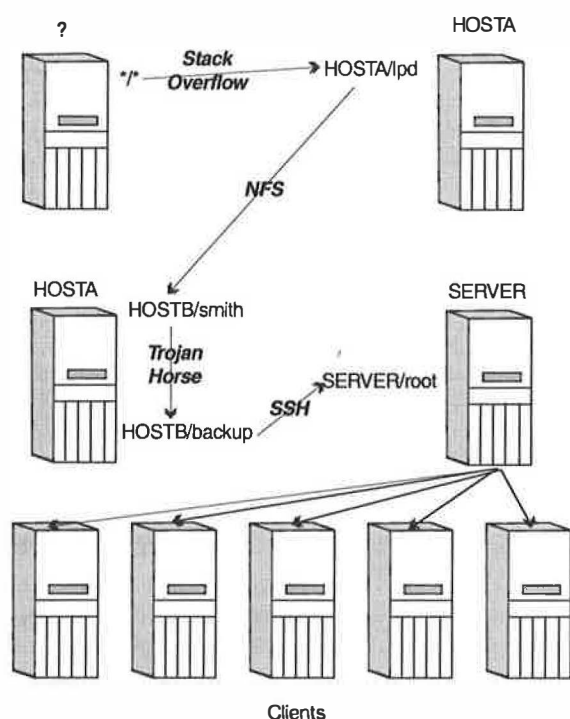


Figure 3: Vulnerability chain.

User Datagram Protocols. The original Expert Fault Manager implementation limited communication to small packets, and sent information as simple ASCII. Packet loss and security wasn't addressed. However, as larger pieces of information was needed, and the system suffered from deadlocks, a redesign was needed. A decision was made to provide the old

function as a base class, and creating sub-classes that add features.

The original system used UDP because of the need for realtime predictable responses, and the desire to extend the protocol to use multicast and broadcast transports in the future. Three low-level UDP classes are broken into UDP Client, UDP Server, and UDP Common functions. All three are sub-classed to provide reliability and packet assembly and disassembly. These are again sub-classed to provide secure communication, using symmetric keys for encryption. The security used is weak, as keys are distributed manually, and reused for each session.

The communication object class provide a simple send and receive function. A system acting as a relay or filter, which is both a client and server, while avoiding deadlocks, merely needs the PERL fragment in Listing 2.

The wait method always waits on the socket associated with the object to the left of the arrow. Additional objects, corresponding to sockets, may be included. This single event loop, combined with timeouts and the test for pending data, was important in preventing deadlocks.

The communication classes contains options for verifying the reliability of the communication. Errors (i.e., dropped packets) can be purposely created by dropping a percentage of the packets pseudo-randomly. This was used to verify the reliability of the communication subsystem. Other options specify buffer sizes, timeouts, status, and counts of packets sent and received. The higher level routines reassemble packet fragments, and retransmit missing packets. The methods used for the reliability and secure sub-classes are the same as the base class as far as the application is concerned. However, there were several dozen private used in the implementation.

Several protocols were attempted to provide reliable communication. To validate the protocol, a

```
my $client = udpsecclient->new( # act as a client to another server
    port=>$port,
    machine=>$machine,
    security=>$level);

my $server = udpsecserver->new( # set up our own server
    port=>$port,
    security=>$level);

while (1) {
    $server->wait($client); # Wait for activity on either socket
    if ($server->pending) { # Our server gets data
        $client->send($server->receive); # send to other server
    }
    if ($client->pending) { # Our client gets results from server
        $server->send($client->receive); # send back to our client
    }
}
```

Listing 2: Simple client/server model.

Design of Experiment (DoE) methodology was used, varying buffer sizes, size of files, and number of simultaneous connections. Developing the reliable protocol was difficult. While the DoE did not clearly indicate the optimum parameters, a reliable protocol was selected that used a common (and symmetric) send/receive method for clients and servers. The sender breaks up the message into smaller pieces, and sends them out sequentially. The receiver responds with a positive acknowledgment, indicating all packets were received, or a negative acknowledgment, which requests the sender to re-transmit the missing packets. Once a positive acknowledgment from the receiver has been received, the sender transmits an acknowledgment of the acknowledgment and changes states. This same method is used by both sides to communicate. Therefore the client requests information from a server, a minimum of 6 packets are transmitted, and more if packets are lost. The primary difference between the client and the server classes was the states: a server is finished after sending information, and the client is finished after receiving information. Caching was easily added, with answers based on the combination of the object and the method. The results are saved if there were no errors.

Functionality

The system can potentially accept vulnerabilities from any source, but a parser must be built to extract the vulnerability type and account information. The system, as currently implemented, performs the following vulnerability checks:

```
Trojan Horse
NFS
RLOGIN/SSH
Output of Alec Muffet's CRACK program
Missing security-related patches
```

The first three operate on account objects (once for each account), while the last two operate on a host basis (once for each host).

The Trojan horse program, uploaded to the agent, finds the shell of the user and examines the appropriate start-up files. It parses the files, and keeps track of files that are sources, as well as the value of variables. When searchpaths are specified or modified, the values of these variables are used to determine the potential searchpath. If branches are taken, the program assumes both paths are used, so the searchpath examined by the program is the superset of the actual searchpath. The program, for safety, doesn't evaluate commands within back-quotes. Instead, a predetermined set of commands are evaluated once on each host, as specified by the program, and if one of the users has this string in their searchpath, the pre-determined value is inserted into the string. If it is not known, it is ignored. Consider the following C shell fragment:

```
if ( -f /local $a)
    localpath = ( /local/'arch'/bin )
else
    localpath = ( /usr/'arch'/bin \
                /usr/local/'arch'/bin )
endif
set path = ( $localpath $path )
```

All three directories will be examined in addition to the default value of the searchpath, assuming the directories exist. The software also detects recursive loops, and handles them to a depth of two, and aborts if a loop is detected. Using this, it is possible to get a searchpath for each account. This, in turn, is used to measure the potential for Trojan Horse attacks on all accounts. Currently the system only examines the permissions of the directories, as well as the permission of the parent directories, and symbolic links. The permission of the files inside the directory are not examined, unlike the author's Trojan checking program.

The algorithm asks the agent to extract all of the user and group information from the host, if necessary. Then it creates a set of account objects matching each found. Originally, each group and account was asked individually, but a new dispatch method was created to retrieve all of the account information in a single query for efficiency.

When one account is selected for the vulnerability check, the IW module asks the agent to get the searchpath, which returns a list of directories by name. The IW module creates instances of these objects as needed in its own database, and asks the agent to list who has write permission for each of the directories (if this has not been asked before). Because each object has a name that is unique, and also specifies the host responsible for the information, instances of objects are externalized, and the object name and method is passed to the agent in plain text, and the results is also in plain text, fragmented and encrypted according to the communication object class. By examining each directory for user-, group- and world-write permission, the agent can return a list of accounts that have the ability to write to the directory specified. A special type of object is needed to correspond to missing directories. This checks for the potential of someone being able to create a directory in the future.

When a Trojan Horse is found, one or more vulnerabilities are created that specifies the victim and the potential attacker(s), as well as the file or directory that caused the problem. In the case of a vulnerability by group-write permission, multiple vulnerabilities are created, listing everyone in the group as a unique attacker. If a directory is world-writable, or the user allows the current directory to be in the searchpath, the attacker is the wildcard account on this machine.

UID's are used when a directory is owned by an account that no longer exists (i.e., has no name associated to it.)

NFS

When an account is examined for NFS vulnerability, the algorithm determines if the file is on a NFS client or a NFS server. Examined on a client, the "attacker" is the root account on the server where the directory is exported, as well as the root account on the client. However, on a server, the system examines the export list, and the netgroup information, and specifies all of the accounts that have the ability to modify the files. In other words, when examining "smith" on "pluto", if "smith" on "neptune" has write permission, then "root" on "neptune" does as well. The system also examines the NFS server if the client port numbers must be less than 1024, indicating a privileged account. If not, then the attacking account is the wildcard account "anyone" on "neptune". If the directory is exported to the world, the attacker is identified as "anyone" on "anyhost," indicating that anyone on any machine that can access the system can break into the account.

RLOGIN

The agent examines the system configuration for /etc/hosts.equiv and \$HOME/.rhosts as well as the SSH equivalent files to determine which counts have access to the selected account. NIS netgroups and "+" in the .rhosts file are understood, and appropriate vulnerabilities are created.

CRACK

The system parses a file generated by CRACK, and looks for a matching hostname. When found, it creates a vulnerability between the "anybody" on this host to the account whose password was guessed.

Checking System Patches

The IW module first reads a series of files that contain a one-line summary per file associated with a patch. The OS type, revision, architecture, file path, patch ID, size, and MD5 value of each file is specified. This file is created automatically for Solaris systems using a shell script that weekly retrieving the current patch status. The IW module, while reading this file, creates Patch, File, OS and Signature objects. These must be separate objects because a file may have multiple signatures depending on the OS, revision, and patch. Also created is a FileState and PatchState object which corresponds to the actual state of a file and patch on a particular host, instead of the generalized information, valid for all systems of the same revision. In other words, PatchState and FileState have an association with a specific host.

The system gets the list of files for each patch and examines the signature of each file if it has permission. Five results are possible: (1) correct revision, (2) wrong revision, (3) file does not exist, and (4) insufficient privileges to read file and (5) system cannot execute the external MD5 program to check

signature. The system can form a conclusion based on partial information. If it cannot read one file of a patch, and a second is the wrong revision, it concludes the patch has not been applied. If it is uncertain because it cannot read all of the files, but the rest are correct, it indicates the patch might be applied. Therefore determining if patches have been applied is accurate even if someone replaced a file after a patch has been applied, or modified any of the utilities (i.e., showrev). Missing patches create different vulnerabilities, depending on permissions of the un-patched file. This is simplistic, and uses the *set-uid* permissions and owner information to identify the attacked account. If the file is a library, the attacked account is considered to be the *root* account. A future version should use an internal MD5 checksum utility to prevent tampering with the executable.

Using the GUI, it is possible for the security officer to see the state of the patch and associated files. If desired, new patches can be uploaded to the system, and applied.

User Interface

The GUI (Figure 4) presents the information in three sections, General operations, host-specific operations, and account-specific operations. The general commands are a series of buttons that print out summaries, load the patch database, dump the database, and interface to other applications for advanced analysis.

The host specific operations include the following actions:

- Uploading the PERL modules.
- Querying the revision of the current modules
- Fetching the user and group information
- Listing all of the UID's on a host
- Listing all of the accounts on a host.
- Scanning the output of CRACK, merging new vulnerabilities
- Scanning the system for missing patches
- Displaying the results of the patch analysis

Displaying the UID or accounts presents the information in a scrolling list below, which can be used for the account-specific operations. Typically accounts are displayed in the scrolling list, and one or more accounts can be selected, and the account-specific methods can be used. Operations iterate over each item selected. If no account is selected, the account in the Argument: location is used. Typically the account that is of primary interest is copied and pasted into this for convenience.

The account-specific methods include:

- Trojan – Perform a Trojan Horse scan on the selected account
- NFS – Perform a NFS vulnerability check
- RLOGIN – Perform checks on rlogin/rsh/rcp/ssh
- Attacker – shows everyone who can attack the selected account

- Attackee – shows everyone the selected account can attack
- Show Vulnerabilities – shows account-specific vulnerabilities

Dispatcher and objects

The dispatch system was reused from an earlier project, using a version of PERL that did not support OO. The earlier project used “pseudo-object-oriented” techniques. That is, the dispatcher used strings to identify objects, methods and parameters. Based on its table, it would select a protocol, host and port and communicate with the remote system using the communication object classes. This included methods for query/response, uploading files and patches, and asking the remote agent to evaluate and execute PERL code. There was a weak correlation between these pseudo-objects and the object class used by the IW module.

Problems Encountered

There were four significant problems with the implementation:

- This prototype evolved from an earlier system, and the object classes were weakly integrated with the remote execution of commands. Agents and their methods could have multiple states. Methods could be undefined or out of date. Queries could be pending, obsolete, or

never asked. Answers could be cached on the agent, or in the dispatch, or integrated into the database. Timeouts could occur anywhere. A redesign with a unified view of the remote information is needed.

- Secondly, retrieving the information was often piecemeal. Sometimes objects were created merely as collection objects and navigation objects. Accounts were created when vulnerabilities were found, and attributes and associations about these objects would often be left undefined. Therefore a large part of the code is testing for the existence of information, and performing queries to fill in missing information if needed.
- Any distributed system is unreliable and asynchronous. Therefore any remote query could fail, and failures had to be handled. A better design would have asynchronous gathering of information, and well-defined algorithms driven by an expert system, reporting on problems.
- The design of the GUI allowed gathering and traversal of the information in any order desired. There was no pre-determined order in the information gathering process. This provided flexibility, but added to the complexity.

These problems caused the code to be bulky and inelegant. Everything worked, but similar code had to be inserted in multiple places.

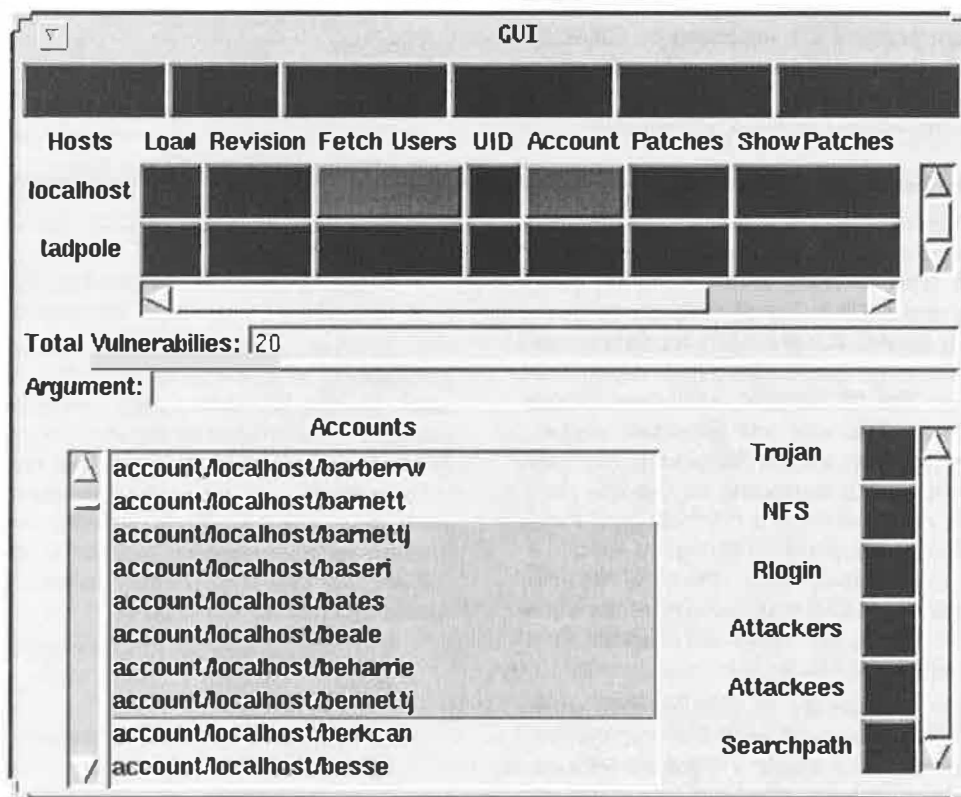


Figure 4: NOOSE GUI.

The second biggest difficulty was the development of the secure, reliable datagram-based protocol. Developing distinct classes between the server and client was essential, but finding the common methods that supported three types of communication (raw, reliable, secure) was difficult. A minor change in the algorithm could easily cause a deadlock.

Conclusions

As in our earlier project [13], an important element to these algorithms is the ability to traverse data structures using associations between objects. This allows algorithms to use the context surrounding objects while making decisions about single objects. This also stimulates new algorithm development, while reusing existing code. In some cases, the objects needed few attributes, as it was used primarily as a navigation point in the data structure.

The system has reasonable performance. Large servers with 2000 accounts could have each account examined in about 30 minutes. A large proportion of this time is believed to be associated with auto mounting the directories found in users search paths.

Because much of the software required handling missing data, and undefined values, a multi-threaded, asynchronous programming model is needed, combined with an object model that unifies the data structures of the agent and system.

The three-way handshake for reliable datagram communication is important for deadlock protection and obvious in hindsight. The symmetric nature allowed code reuse, and therefore simplified the object classes. Using this, it was possible to transmit patch clusters of 70 megabytes. Occasionally timeouts occurred where both sides were waiting, but the system typically recovered. Creating communication object classes designed for sub-classing allows future extensions while retaining the same API.

Future Directions

The communication classes can be extended in several ways. A better key distribution system can be inserted, as well as alternate encryption algorithms. Also, a subclass can be created that combines multiple lightweight messages in a single authenticated/encrypted packet. A mechanism for proxy agents can be added, as well as a means to locate agents by broadcast. Threaded asynchronous communication would simplify code development, as would uniform methods for testing and retrieving remote attributes.

Objects are currently stored in memory only, or dumped to an external file for analysis. An ASCII representation of the database is about 1 Megabyte in size for 2000 vulnerabilities. A persistent database is desirable, especially to a relational database.

Alternate mechanisms of viewing and analyzing vulnerabilities is desired. Several simple algorithms for categorizing vulnerabilities are suggested:

- Identify files with the largest number of associated vulnerabilities.
- Counting the number of paths between any two accounts.
- Identify all of the accounts that can potentially break into a selected account.
- Identify the worst case result of a single compromised account.

The author feels the concepts can be used for advanced policy management systems, as security can be measured more accurately, and relationships can be constructed between files and services.

Summary

The author strongly feels that object modeling is essential to writing next-generation security algorithms, allowing a single database to be used for diverse algorithms. This single database merges together the base functionality of host-based scanners, network-based scanners, file tampering and intelligent patch management systems. The author hopes the object model will be useful to others developing similar applications.

The author believes that this implementation shows several unique traits. The implementation simplifies prototyping new algorithms, and allows reusing data for multiple applications. Adding the patch management software was simpler than the core communication classes. The agent structure simplifies support. The object class for communication can be extended in many ways. The use of the vulnerability and account object class provides a simple and elegant, yet powerful way to integrate information from multiple sources, as we feel there is great potential and flexibility to this mechanism.

Author Information

Bruce Barnett graduated from RPI in 1973. He is currently a Computer Scientist doing research at General Electric's Corporate Research and Development Center, PO Box 8, Schenectady, NY, 12309. His research covers traffic analysis, expert systems, real-time video multicast, and security systems. His electronic address is barnett@crd.ge.com.

References

- [1] Robert W. Baldwin, *Kuang: Rule-based security checking*, Documentation in <ftp://ftp.cert.org/pub/tools/cops/1.04/cops.tar>.
- [2] Dan Farmer & Eugene H. Spafford, "The Cops Security Checker System," *USENIX, Summer 1990*.
- [3] D. Farmer and W. Venema, "Security administrator's tool for analyzing networks," <http://www.fish.com/zen/satan/satan.html>.
- [4] Gene Kim and E. H. Spafford, *The design of a system integrity monitor: Tripwire*, Technical Report CSD-TR-93-071, Department of

- Computer Sciences, Purdue University, West Lafayette, Indiana, November 1993.
- [5] Dan Zerkle and Karl Levitt, "NetKuang – A Multi-Host Configuration Vulnerability Checker," *USENIX 1996*.
 - [6] Bruce Barnett, <ftp://coast.cs.purdue.edu/pub/tools/unix/trojan/trojan.pl>.
 - [7] Doug Schales, "Tiger," <ftp://coast.cs.purdue.edu/pub/tools/unix/tiger>.
 - [8] Internet Security Systems, *Internet Scanner and System scanner*, <http://www.iss.net/>.
 - [9] Diego Zamboni, *SAINT: A Security Analysis Integration Tool*, <ftp://coast.cs.purdue.edu/pub/doc/tools/SAINT.ps.gz>.
 - [10] Diego Zamboni, *New COPS Analysis and Report*, <ftp://coast.cs.purdue.edu/pub/tools/unix/carp-ncarp>.
 - [11] *Merlin*, <http://ciac.llnl.gov/ciac/ToolsMerlin.html>.
 - [12] *SPI – Security Profile Inspector*, <http://ciac.llnl.gov/cstc/spi/spiwnt/spiv20.html>.
 - [13] Bruce Barnett, Andrew Crapo, "An Expert Fault Manager using an Object Meta-Model", *Proceedings 20th Conference on Local Computing Networks*, Minneapolis, MN, Oct 1995.
 - [14] Dai Nha Wu, Bruce Barnett, "Vulnerability Assessment and Intrusion Detection with Dynamic Software Agents," *Ninth Annual Software Technology Conference*, Salt Lake City, Utah, May 1997.

THE USENIX ASSOCIATION

Since 1975, the USENIX Association has brought together the community of developers, programmers, system administrators, and architects working on the cutting edge of the computing world. USENIX conferences have become the essential meeting grounds for the presentation and discussion of the most advanced information on new developments in all aspects of advanced computing systems. USENIX and its members are dedicated to:

- problem-solving with a practical bias
- fostering innovation and research that works
- communicating rapidly the results of both research and innovation
- providing a neutral forum for the exercise of critical thought and the airing of technical issues

SAGE, the System Administrators Guild

The System Administrators Guild, a Special Technical Group within the USENIX Association, is dedicated to the recognition and advancement of system administration as a profession. To join SAGE, you must be a member of USENIX.

Member Benefits:

- Free subscription to *login*., the Association's magazine, published eight times a year, featuring technical articles, system administration tips and techniques, practical columns on Perl, Java, Tcl/Tk, and Open Source, book and software reviews, summaries of sessions at USENIX conferences, and Standards Reports from the USENIX representative and others on various ANSI, IEEE, and ISO standards efforts.
- Access to *login* on the USENIX Web site.
- Access to papers from the USENIX Conferences and Symposia, starting with 1993, on the USENIX Web site.
- Discounts on registration fees for the annual, multi-topic technical conference, the System Administration Conference (LISA), and the various single-topic symposia addressing topics such as security, Linux, Internet technologies and systems, operating systems, and Windows—as many as twelve technical meetings every year.
- Discounts on the purchase of proceedings and CD-ROMs from USENIX conferences and symposia and other technical publications.
- The right to vote on matters affecting the Association, its bylaws, and election of its directors and officers.
- Savings on a variety of products, books, software, and periodicals: see <http://www.usenix.org/membership/specialdisc.html> for details.

Supporting Members of the USENIX Association:

Addison-Wesley
Earthlink Network
Edgix
Interhack Corporation
Interliant
Lucent Technologies

Microsoft Research
Motorola Australia Software Centre
Nimrod AS
O'Reilly & Associates Inc.
Sams Publishing
Sendmail, Inc.

Smart Storage, Inc.
Sun Microsystems, Inc.
Sybase, Inc.
Syntax, Inc.
Taos: The Sys Admin Company
UUNET Technologies, Inc.

Supporting Members of SAGE:

Collective Technologies
Deer Run Associates
Electric Lightwave, Inc.
ESM Services, Inc.
GNAC, Inc.
Mentor Graphics Corp.

Microsoft Research
Motorola Australia Software Centre
New Riders Press
O'Reilly & Associates Inc.
Remedy Corporation

RIPE NCC
Sams Publishing
SysAdmin Magazine
Taos: The Sys Admin Company
Unix Guru Universe

For more information about membership, conferences, or publications,
see <http://www.usenix.org/>
or contact:

USENIX Association, 2560 Ninth Street, Suite 215, Berkeley, CA 94710 USA.
Phone: 510-528-8649. Fax: 510-548-5738. Email: office@usenix.org.

